Real-Time Concurrent Processes

Final Technical Report

31 March 2000

Contract Number F49620-97-C-0008

Prepared For

U.S. Air Force Office of Scientific Research
Directorate of Mathematics and Space Sciences
800 North Randolph Street
Arlington, VA, 22203-1977

Prepared By

Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

Real-Time Concurrent Processes
Final Technical Report

Contract Number F49620-97-C-0008

Prepared for

Capt. Alex Kilpatrick
Program Manager, Software and Systems
U.S. Air Force Office of Scientific Research
800 North Randolph Street
Arlington, VA, 22203-1977
freeman.kilpatrick@afosr.af.mil
(703) 696-6565

Prepared by

Steve Vestal
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418
steve.vestal@honeywell.com
(612) 951-7049

REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-00-

0221

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3.
	27 March 2000	Final Technical Report 1 Apr 97 to 31 Dec 99

4. TITLE AND SUBTITLE
Real-Time Concurrent Processes

5. FUNDING NUMBERS
F49620-97-C-0008
2304/FS

6. AUTHOR(S)
Steve Vestal

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
801 N. Randolph St, Rm 732
Arlington, VA 22203-1977

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-97-C-0008

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Discrete event concurrent process models are widely used to model control flow within and interactions between concurrent activities. Classical discrete event concurrent process models do not deal with resource allocation and scheduling or data variables, which limits their usefulness for real-time systems and makes it awkward to model some implementation details. Classical preemptive scheduling models do not deal with complex task sequencing and interaction, which limits their usefulness for describing distributed systems and implementation details. Discrete time models have been developed for real-time scheduling of concurrent processes [10, 5, 4. 11], and some work has been done on dense time real-time process algebras [3, 6]. This report describes the use of dense time linear hybrid automata models to perform schedulability analysis and to verify implementation code.

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Executive Summary

This section provides an executive summary of the work performed during the period April 1997 through March 2000. This section also outlines future work activities that may result in additional significant advances and benefits. A more extensive summary appears in an attached paper[15], various technical details are described in attached papers[12, 13, 14].

The first goal of the work described in this report was to analyze the schedulability of real-time systems that cannot be easily modeled using traditional scheduling theory. Traditional real-time task models cannot easily handle variability and uncertainty in clock and computation and communication times, synchronizations (rendezvous) between tasks, remote procedure calls, anomalous scheduling in distributed systems, dynamic reconfiguration and reallocation, end-to-end deadlines, and timeouts and other error handling behaviors.

The second goal was to verify software implementations of systems. Task schedulers and communications protocols are reactive components that respond to events like interrupts, service calls, task completions, error detections, etc. We would like to model important implementation details such as control logic and data variables in the code. We would like the mapping between model and code to be clear and simple to better assure that the model really does describe the implementation.

Discrete event concurrent process models are widely used to model control flow within and interactions between concurrent activities. Classical discrete event concurrent process models do not deal with resource allocation and scheduling or data variables, which limits their usefulness for real-time systems and makes it awkward to model some implementation details. Classical preemptive scheduling models do not deal with complex task sequencing and interaction, which limits their usefulness for describing distributed systems and implementation details. Discrete time models have been developed for real-time scheduling of concurrent processes[10, 5, 4, 11], and some work has been done on dense time real-time process algebras[3, 6]. This report describes the use of dense time linear hybrid automata models to perform schedulability analysis and to verify implementation code.

The first problem we faced was the modeling of resource allocation and scheduling behaviors using hybrid automata. The applicability in principle of hybrid automata to the scheduling problem was already known[2], but the examples published in the research literature did not deal with some important practical problems. We wanted a model that would admit a variety of complex allocation as well as scheduling algorithms, e.g. load balancing, dynamic priorities. We wanted to be able to change the allocation and scheduling algorithms easily without changing the models of the real-time tasks themselves. We wanted to minimize the number of states and variables added to model allocation and scheduling. We found it most general and efficient to extend the definition of hybrid automata to include resource allocation and scheduling semantics rather than try to model the scheduling function as a hybrid automaton. The result is a very powerful model that admits distributed heterogeneous systems with a great variety of dynamic reallocation and scheduling algorithms, and our experience confirms that hybrid automata models are well-suited to this domain. This is described in detail in attached papers[12, 13].

We use integration variables to record the accumulated compute time of tasks in preemptively scheduled systems. Allowing integration variables is known to make the reachability problem undecidable[9, 7]. We were curious about whether analysis of real-time allocation and scheduling in distributed heterogeneous systems is itself a fundamentally difficult problem, or if general linear

hybrid automata are more powerful than is really necessary for this problem. We were able to show that the reachability problem becomes decidable when some simple pragmatic restrictions are placed on the model. The proof is by reduction to a discrete time finite state automaton, which establishes an equivalence between dense time and discrete time models for this problem with these restrictions. However, there remain some interesting open questions in this area. This is described in greater detail in an attached paper[12].

The second problem we faced was the computational difficulty of performing a reachability analysis. We began our work using an existing linear hybrid automata analysis tool, HyTech[8], but found ourselves limited to very small models. We developed and implemented a new reachability method that was significantly faster, more numerically robust, and used less memory. However, our prototype tool allows only constant rates (not rate ranges) and does not provide parametric analysis. We further increased the size of model we could analyze by applying some results from traditional scheduling theory to condition the models, and by using a simple partial order reduction technique. These results are described in detail in an attached paper[13].

Using this new reachability procedure we were able to accomplish one of our goals: the modeling and verification of a piece of real-time software. We developed a hybrid automata model for that portion of the MetaH real-time executive that implements uniprocessor task scheduling, time partitioning and error handling[1]. We successfully analyzed these models, uncovering several implementation defects in the process. There are limits on the degree of assurance that can be provided, but in our judgement the approach may be significantly more thorough and significantly less expensive that traditional testing methods. This result suggests the technology has reached the threshold of practical utility for the verification of small amounts of software of a particular type. These results are described in detail in an attached paper[14].

There are two major areas of activity that may yield significant future benefits. These are discussed in somewhat more detail in attached papers[13, 14].

Our results to date are, in our judgement, adequate to verify certain pieces of software of real-world size and complexity, but not yet adequate to verify large pieces of software or perform a schedulability analysis for a distributed system of useful size. There are, however, several approaches that might increase by another two orders of magnitude or more the size of problem that can be analyzed. Our preliminary work has shown how conditioning a model by conservatively changing the numeric parameters can significantly reduce the complexity of analysis; and how the use of a partial order reduction method adapted to hybrid automata can reduce the complexity of analysis. We and others have explored methods that approximate a set of polyhedra by a single containing polyhedra. Success to date has been limited, our experience has been that it is difficult to achieve significant reductions in solution complexity while at the same time retaining sufficient accuracy. However, we believe approximation methods can be refined to achieve this. We also need to extend our methods to support ranges of variable rates, to provide parametric analysis, and to make more convenient a variety of practical tasks such as specification and debugging and visualization.

The methods we have developed are likely to be applicable to domains other than just hard real-time scheduling. Hybrid automata models have been widely discussed in the research literature for certain types of feed-back control problems, our improved reachability analysis methods may enable practical application in some cases. Our basic approach of adding domain semantics might work for other applications whose models include many concurrent processes. Our approach to scheduling

is to categorize general and easily computed policies applied to a class of related problems, rather than attempt to synthesize a specific controller for a specific problem. Planning and scheduling, for example, has traditionally relied on computationally intensive off-line generation of static plans that are not dynamically adaptable to contingencies that occur during execution. This domain might benefit from an approach similar to that used for real-time scheduling, where a very efficient and analytically verifiable (if not always optimal) policy is used to rapidly make decisions on-line. Another new domain of applicability is software testing, where points selected from the reachable set of regions produced from a model could be used for automatic test generation.

Bibliography

[1] *MetaH User's Guide*, Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN, `www.htc.honeywell.com/metah`.

[2] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.

[3] Patrice Brémond-Grégoire and Insup Lee, "A Process Algebra of Communicating Shared Resources with Dense Time and Priorities," University of Pennsylvania Department of Computer Science Technical Report MS-CIS-95-08, June 1996.

[4] S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, "Computing Quantitative Characteristics of Finite-State Real-Time Systems," *Real-Time Systems Symposium*, December 1994.

[5] Andre N. Fredette and Rance Cleaveland, "RSTL: A Language for Real-Time Schedulability Analysis," *Proceedings of the Real-Time Systems Symposium*, December 1993.

[6] Andre N. Fredette, *A Generalized Approach to the Analysis of Real-Time Computer Systems*, Ph.D. Dissertation, North Carolina State University, March 1993.

[7] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri and Pravin Varaiya, "What's Decideable About Hybrid Automata?" *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995.

[8] Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, "HyTech: The Next Generation," *Real-Time Systems Symposium*, December 1995.

[9] Y. Kesten, A. Pnueli, J. Sifakis and S. Yovine, "Integration Graphs: A Class of Decideable Hybrid Systems," in R. L. Grossman, A. Nerode, A. P. Ravn and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, Springer-Verlag, 1993.

[10] Insup Lee, Patrice Brémond-Grégoire and Richard Gerber, "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," Department of Computer Science, University of Pennsylvania.

[11] Jin Yang, Aloysius K. Mok and Farn Wang, "Symbolic Model Checking for Event-Driven Real-Time Systems," *ACM Transactions on Programming Languages and Systems*, v19, n2, March 1997.

[12] Steve Vestal, "Linear Hybrid Automata Modeling of Dyanamic Real-Time Allocation and Scheduling in Distributed Heterogeneous Systems," Honeywell Technology Center, 1998.

[13] Steve Vestal, "A New Linear Hybrid Automata Reachability Procedure," Honeywell Technology Center, 1999.

[14] Steve Vestal, "Formal Verification of the MetaH Executive Using Linear Hybrid Automata," to appear *Real-Time Applications Symposium*, May 2000.

[15] Steve Vestal, "Modeling and Verification of Real-Time Software Using Extended Linear Hybrid Automata," to appear *NASA Langley Formal Methods Workshop*, June 2000.

Modeling and Verification of Real-Time Software Using Extended Linear Hybrid Automata

Steve Vestal
steve.vestal@honeywell.com
Honeywell Technology Center
Minneapolis, MN 55418*

Abstract

Linear hybrid automata are finite state automata augmented with real-valued variables. Transitions between discrete states may be conditional on the values of these variables and may assign new values to variables. These variables can be used to model real time and accumulated task compute time as well as program variables. Although it is possible to encode preemptive fixed priority scheduling using existing linear hybrid automata models, we found it more general and efficient to extend the model slightly to include resource allocation and scheduling semantics. Under reasonable pragmatic restrictions for this problem domain, the reachability problem is decidable. The proof of this establishes an equivalence between dense time and discrete time models given these restrictions. We next developed a new reachability algorithm that significantly increased the size of problem we could analyze, based on benchmarking exercises we carried out using randomly generated real-time uniprocessor workloads. Finally, we assessed the practical applicability of these new methods by generating and analyzing hybrid automata models for the core scheduling modules of an existing real-time executive. This exercise demonstrated the applicability of the technology to real-world problems, detecting several errors in the executive code in the process. We discuss some of the strengths and limitations of these methods and possible future developments that might address some of the limitations.

1 Introduction

The first goal of the work described in this paper was to analyze the schedulability of real-time systems that cannot be easily modeled using traditional scheduling theory. Traditional real-time task models cannot easily handle variability and uncertainty in clock and computation and communication times, synchronizations (rendezvous) between tasks, remote procedure calls, anomalous scheduling in distributed systems, dynamic reconfiguration and reallocation, end-to-end deadlines, and timeouts and other error handling behaviors.

The second goal was to verify software implementations of systems. Task schedulers and communications protocols are reactive components that respond to events like interrupts, service calls, task completions, error detections, etc. We would like to model important implementation details such as control logic and data variables in the code. We would like the mapping between model and code to be clear and simple to better assure that the model really does describe the implementation.

Discrete event concurrent process models are widely used to model control flow within and interactions between concurrent activities. Classical discrete event concurrent process models do not deal with resource allocation and scheduling or data variables, which limits their usefulness for real-time systems and makes it awkward to model some implementation details. Classical preemptive scheduling models do not deal well with complex task sequencing and interaction, which limits their usefulness for describing distributed systems and implementation details. Discrete time models have been developed for real-time scheduling of concurrent processes[23, 13, 11, 31], and some work has been done on dense time real-time process algebras[10, 14]. This paper describes the use of dense time linear hybrid automata models to perform schedulability analysis and to verify implementation code.

The first problem we faced was the modeling of resource allocation and scheduling behaviors using hybrid automata. The applicability in principle of hy-

*This work has been supported by the Air Force Office of Scientific Research under contract F49620-97-C-0008. This paper appears in the proceedings of the NASA Langley Formal Methods Workshop, June 2000.

brid automata to the scheduling problem was already known[4]. We wanted a model that would admit a variety of complex allocation as well as scheduling algorithms, e.g. load balancing, priority inheritance. We wanted to be able to change the allocation and scheduling algorithms easily without changing the models of the real-time tasks themselves. We wanted to minimize the number of states and variables added to model allocation and scheduling. We found it most general and efficient to extend the definition of hybrid automata to include resource allocation and scheduling semantics rather than try to model the scheduling function as a hybrid automaton.

We use integration variables to record the accumulated compute time of tasks in preemptively scheduled systems. Allowing integration variables is known to make the reachability problem undecidable[22, 17]. We were curious about whether analysis of real-time allocation and scheduling in distributed heterogeneous systems is itself a fundamentally difficult problem, or if general linear hybrid automata are more powerful than is really necessary for this problem. We were able to show that the reachability problem becomes decidable when some simple pragmatic restrictions are placed on the model.

The second problem we faced was the computational difficulty of performing a reachability analysis. We began our work using an existing linear hybrid automata analysis tool, HyTech[18], but found ourselves limited to very small models. We developed and implemented a new reachability method that was significantly faster, more numerically robust, and used less memory. However, our prototype tool allows only constant rates (not rate ranges) and does not provide parametric analysis.

Using this new reachability procedure we were able to accomplish one of our goals: the modeling and verification of a piece of real-time software. We developed a hybrid automata model for that portion of the MetaH real-time executive that implements uniprocessor task scheduling, time partitioning and error handling[1]. We successfully analyzed these models, uncovering several implementation defects in the process. There are limits on the degree of assurance that can be provided, but in our judgement the approach may be significantly more thorough and significantly less expensive that traditional testing methods. This result suggests the technology has reached the threshold of practical utility for the verification of small amounts of software of a particular type.

However, we do not believe existing reachability methods are adequate yet for schedulability analysis of real systems. In our judgement, we would need to be able to analyze systems having a few dozen tasks on a few processors in order for the technology to begin finding use in this area. We discuss approaches that might lead to such improvements.

2 Resourceful Hybrid Automata

A hybrid automaton is a finite state machine augmented with a set of real-valued variables and a set of propositions about the values of those variables. Figure 1 shows an example of a hybrid automaton whose discrete states are `preempted`, `executing` and `waiting`; and whose real-valued variables are c and t. `Waiting` is marked as the initial discrete state, and c and t are assumed to be initially zero.

Each of the discrete states has an associated set of differential equations, e.g. $\dot{c} = 0$ and $\dot{t} = 1$ for the discrete state `preempted`. While the automaton is in a discrete state, the continuous variables change at the rates specified for that state.

Edges may be labeled with guards involving continuous variables, and a discrete transition can only occur when the values of the continuous variables satisfy the guard. When a discrete transition does occur, designated continuous variables can be set to designated values as specified by assignments labeling that edge.

A discrete state may also be annotated with an invariant constraint to assure progress. Some discrete transition must be taken from a state before that state's invariant becomes false. For example, the hybrid automaton in Figure 1 must transition out of state `computing` before the value of c exceeds 100.

The hybrid automata of interest to us are called linear hybrid automata because the invariants, guards and assignments are all expressed as sets of linear constraints. The differential equations governing the continuous dynamics in a particular discrete state are restricted to the form $\dot{x} \in [l, u]$ where $[l, u]$ is a fixed constant interval (our current prototype tool is further restricted to a singleton rate, $\dot{x} = [l, l]$).

We want to verify assertions about the behavior of a hybrid automaton. Although it is possible in general to check temporal logic assertions[4], we make do by annotating discrete states and edges with sets of linear constraints labeled as assertions. These constraints must be true whenever the system is in a discrete state or whenever a transition occurs over an edge.

The cross-product construction used to compose concurrent finite state processes can be extended in a fairly straight-forward way to systems of hybrid automata. The invariant and assertion associated with a discrete system state are the conjunction of the invari-

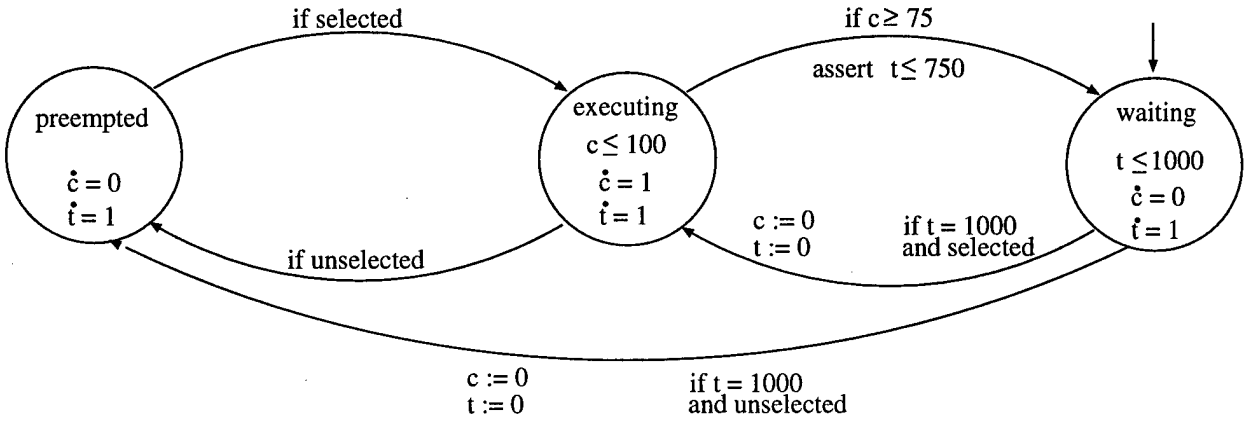


Figure 1: A Hybrid Automata Model of a Preemptively Scheduled Task

ants and assertions of the individual discrete states. The guards, assertions and assignments of synchronized transitions are the conjunction and union of the guards, assertions and assignments of the individual discrete co-edges. If there is a conflict between the rate assignments of individual discrete states, or a conflict between the variable assignments of co-edges, then the system is considered ill-formed. Note that concurrent hybrid automata may interact through shared real-valued variables as well as by synchronizing their transitions over co-edges.

The application of interest in this paper is the analysis and verification of real-time systems. Figure 1 shows an example of a simple hybrid automata model for a preemptively scheduled, periodically dispatched task. A task is initially waiting for dispatch but may at various times also be executing or preempted. The variable t is used as a timer to control dispatching and to measure deadlines. The variable t is set to 0 at each dispatch (each transition out of the waiting state), and a subsequent dispatch will occur when t reaches 1000. The assertion $t \leq 750$ each time a task transitions from executing to waiting (each time a task completes) models a task deadline of 750 time units. The variable c records accumulated compute time, it is reset at each dispatch and increases only when the task is in the computing state. The invariant $c \leq 100$ in the computing state means the task must complete before it receives more than 100 time units of processor service, the guard $c \geq 75$ on the completion transition means the task may complete after it has received 75 time units of processor service (i.e. the task compute time is uncertain and/or variable but always falls in the interval $[75, 100]$).

In this example the edge guards `selected` and `unselected` represent scheduling decisions made at scheduling events (called scheduling points in the real-time literature). These decisions depend on the available resources (processors, busses, etc.) being shared by the tasks. There are several approaches to introduce scheduling semantics into a model having several concurrent tasks.

Scheduling can be introduced using concepts taken from the theory of discrete event control[26]. A concurrent scheduler automaton can be added to the system of tasks. The scheduling points in the task set become synchronization events at which the scheduler automaton can observe the system state and make control decisions. Many high-level concepts from discrete event control theory carry over into this domain, such as the importance of decentralized control and limited observability in distributed systems.

Discrete event control theory provides an approach to synthesize optimal controllers, which in this domain translates to the automatic construction of application-specific scheduling algorithms. However, classical discrete event control theory does not deal with time. The theory has been extended to synthesize nonpreemptive schedulers for timed automata[9, 2], but this excludes preemptively scheduled systems. It is possible to develop scheduling automata by hand using traditional real-time scheduling policies such as preemptive fixed priority. Some examples have been given in the literature, where each distinct ready queue state is modeled as a distinct discrete state of the scheduler automaton[4]. This would allow a very large class of scheduling algorithms to be modeled, but the size of the scheduler automaton may grow combinatorially with the number of tasks.

It is possible to model preemptive fixed priority

scheduling by encoding the ready queue in a variable rather than in a set of discrete states. A queue variable is introduced that will take on only integer values. At each transition where a task i is dispatched, 2^i is added to this queue variable; at each transition where task i completes, 2^i is subtracted. The queue variable can be interpreted as a bit vector whose i^{th} bit is set whenever task i is ready to compute. There is no separate scheduler automaton, the scheduling protocol is modeled using additional guards and states in the task automata. This is the approach we took when we started our work using HyTech. This encodes a specific scheduling protocol into each task model, and adds additional discrete states, variables and guards to the model. It is awkward to model any scheduling policy other than simple preemptive fixed priority.

In the end, we found it simpler and more general to define a slightly extended linear hybrid automata model that includes resource scheduling semantics[28]. The discrete state composition of the task set is performed before any scheduling decisions are made. A scheduling function is then applied to the composed system discrete state to determine the variable rates to be used for that system state. In essence, the composed system discrete state is the ready queue to which the scheduling function is applied, very much analogous to the way run-time scheduling algorithms are applied in an actual real-time system. It is not necessary to have different discrete states for preempted and computing, since this information is now captured in the variable rates. It is not necessary to model a scheduling algorithm as a finite state control automaton added to the system, it is not necessary to encode a specific scheduling semantics into the task automata. One simply codes up a scheduling algorithm in the usual way and links it with the rest of the reachability analysis code. This approach significantly reduces the number of discrete states in the model (from 3^t for our HyTech models to 2^t for our extended models, where t is the number of tasks). This also simplifies the modeling of the desired scheduling discipline. The details of this model and its semantics are recorded elsewhere[28].

3 Decideability

Most traditional real-time schedulability problems are solvable in polynomial time or are NP-complete. However, hybrid automata models that allow multiple rates and integration variables are undecideable[22, 17]. The hybrid automata models we are using are much more powerful than traditional allocation and scheduling models, and most existing tasking and scheduling models can be viewed as special cases of the more general hybrid automata model. This raises the question of whether the schedulability problem for complex interacting tasks that are dynamically allocated in distributed heterogeneous systems is in fact undecideable, or whether models of such systems are decideable special cases of the more powerful linear hybrid automata models.

The undecideability of hybrid automata reachability analysis was proved by reducing the reachability problem for two-counter machines, which is known to be undecideable, to the reachability problem for hybrid automata[22, 17]. The construction used in the proof is fairly straightforward in our slightly extended model and can be accomplished using a single processor. However, a pragmatic real-time system designer would reject the theoretical construction as a bad design because it relies in places on exact equality comparisons between timers and accumulated compute times. In a real system, these would be regarded as race conditions or ill-defined behaviors. The problem becomes decideable given a few simple practical restrictions, which are captured in the following theorem.

Theorem 1 *The reachability problem is decideable for resourceful linear hybrid automata if the following conditions hold.*

- *The set of possible outputs of the scheduling function for each possible system discrete state is finite and enumerable.*
- *For every task activity integrator variable, the rate interval remains fixed between resets of that integrator (i.e. the scheduler does not dynamically reallocate any task activity in mid-execution to a new resource having a different rate for that activity).*
- *For every task activity integrator variable, every edge guard is a set of rectangular constraints of the form $x \in [l, u]$, and either the edge guard has a non-singular interval ($x \in [l, u]$ with $l < u$) or else the rate interval for $\dot{x}$ is non-singular (i.e. system behavior does not depend on exact equality comparisons with exact drift-free clocks or execution rates).*
- *However, we allow as a special exception task activity integrator variables with singular rate interval and singular rectangular edge guards, providing the integrator variable is only reset or stopped or restarted at a transition having at least one edge guard $y \in [m, m]$ with $[m, m]$ and $\dot{y}$ singular (y may but need not be x), and for every such*

singular constraint on that edge $\dot{x} = k\dot{y}$ for some positive integer k (i.e. some types of noninteracting or harmonically interacting behaviors may be modeled exactly).

This result should not be surprising. The ability to test for exact equality is known to add theoretical power to dense time temporal logics[3], and similar restrictions are known to make certain other hybrid automata models decideable[25]. The proof of this theorem, which we provide elsewhere[28], is by reduction to a discrete time finite state automaton.

4 Reachability Analysis

A state of a linear hybrid automaton consists of a discrete part, the discrete state at some time t; and a continuous part, the real values of the variables at time t. It turns out that, although this state space is uncountably infinite, the reachable state space for a given linear hybrid automaton is a subset of the cross-product of the discrete states with a recursively enumerable set of convex polyhedra in $\Re^n$ (where n is the number of variables)[4]. A region of a linear hybrid automaton is a pair consisting of a discrete state and a convex polyhedron, where convex polyhedra can be represented using a finite set of linear constraints. Model checking consists of enumerating the reachable regions for a given linear hybrid automaton and checking to see if they satisfy the assertions.

Figure 2 depicts the basic sequence of operations that, given a starting region (a discrete state and a polyhedron defining a set of possible values for the variables), computes the set of values the variables might take on in that discrete state as time passes; and computes a set of regions reachable by subsequent discrete transitions.

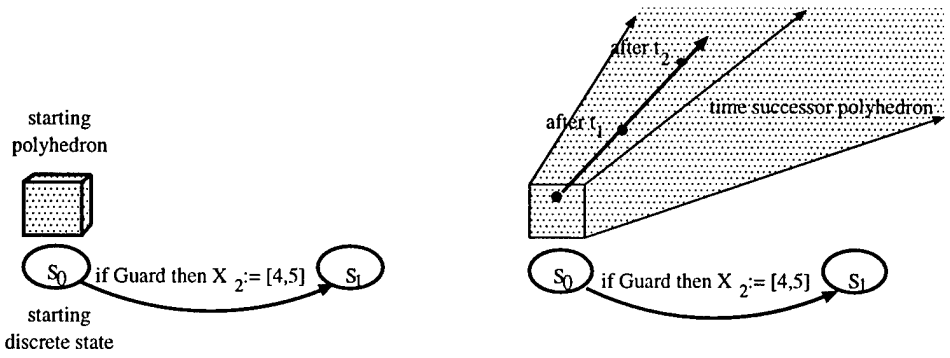
The first step is the computation of the time successor polyhedron from the starting polyhedron (often called the post operation). For each point in the starting polyhedron, the time successor of that point is a line segment beginning at that point whose slope is defined by the variable rates specified for the discrete state. This is the set of variable values that can be reached from a starting point by allowing some amount of time to pass. The time successor of the starting polyhedron is the union of the time successor lines for all points in the starting polyhedron. A basic result of linear hybrid automata theory is that the time successor of any convex polyhedron is itself a convex polyhedron (which in general will be unbounded in certain directions)[4].

The second step is the intersection of the time successor polyhedron with the invariant constraint associated with the discrete state. Polyhedra are easily intersected by taking the union of the set of linear constraints that define the two polyhedra. This is the time successor region that is feasible given the invariant specified for the discrete state.

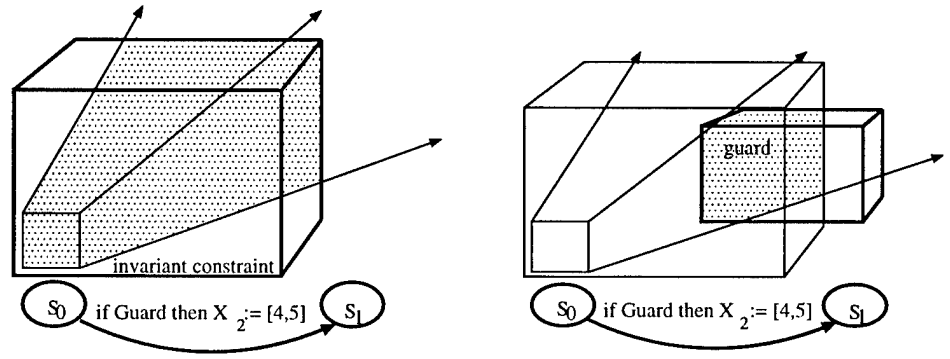
The remaining steps are used to compute new regions reachable from this feasible time successor region by some transition over an edge. For each edge out of the current discrete state, the associated guard is first intersected with the feasible time successor region. This polyhedron, if nonempty, defines the set of all variable values that might exist whenever the discrete transition could occur. Any variable assignments associated with the edge must now be applied to this polyhedron. This is done in two phases. First, a variable to be assigned a new value $x := l$ is unconstrained (often called the free operation). This operation leaves unchanged the relationships between all other variables, i.e. the polyhedron is projected onto the subspace $\Re^{n-1}$ of the remaining variables. This result is then intersected with the constraint $x = l$. This polyhedron, together with the discrete state to which the edge goes, is a new region for which the above steps may be repeated. In general a set of assignments whose right-hand sides are linear formula are allowed, with some restrictions. The variables to be assigned are unconstrained and the resulting polyhedra are then intersected with the appropriate linear constraints in some order. With care, fairly complex sequences of assignments to program variables can be modeled on a single edge[30].

The overall method begins at the initial region of a hybrid automaton. The operations described above are applied to enumerate feasible time successor regions and the new regions reachable from these via discrete transitions. As new regions are enumerated, they must be checked to see if they have been visited before (otherwise the method will not terminate even when there are a finite number of regions). This is done by comparing the discrete states of regions for equality, and by checking to see if the new polyhedron is contained in the polyhedron of a previously visited region.

The earliest reachability tool of which we are aware, HyTech, represented polyhedra as finite sets of linear constraints[4]. Operations on polyhedra used quantifier elimination, a method to manipulate and make decisions about systems of linear constraints in which some of the variables are existentially quantified. Subsequent tools, Polka and a later version of HyTech, used a pair of representations: the traditional system of linear constraints together with polyhedra gener-

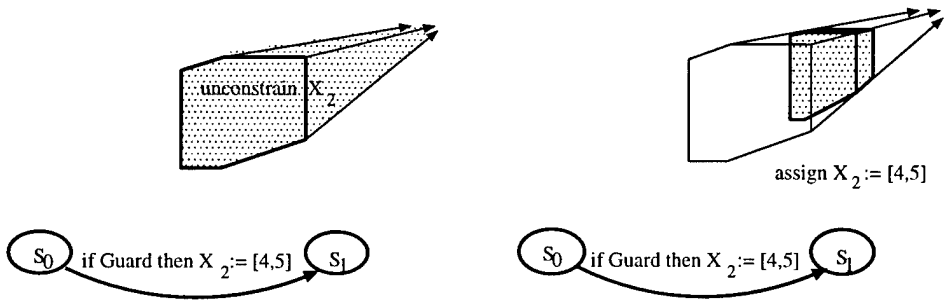


Step 1: Time Successor



Step 2: Intersection with Invariant

Step 3: Intersection with Guard



Step 4: Unconstrain Assigned Variable

Step 5: Intersect with Assignment Constraint

Figure 2: Hybrid Automata Reach Forward Operations

ators consisting of sets of vertices and rays[16, 18]. Different operations required during reachability are more convenient in the different representations, and methods are used to convert between the two as needed.

Both of these methods are subject to the theoretical risk that some polyhedra operations may require a combinatorial amount of time. Another potential performance problem occurs when the reachable discrete state space is completely enumerated first followed by

an enumeration of the polyhedra. This might result in enumerating discrete states that are actually not reachable due to edge guards involving the continuous variables. Finally, in our experiments we found that a significant fraction of a set of benchmark schedulability problems we tried to solve using HyTech resulted in numeric overflow errors.

We developed a new set of algorithms for the polyhedra operations used during reachability analysis and implemented a prototype on-the-fly reachability analysis library. Our prototype operates on lists of linear constraints of the form $l \leq e \leq u$ where l and u are fixed constant integer bounds and $e = c_1x_1 + c_2x_2 + ...$ is a linear formula with fixed constant integer coefficients. Our current algorithms restrict variable rates to be fixed scalar constants, $\dot{x} = i$ rather than the more general $\dot{x} \in [l, u]$.

We convert a polyhedron P into $\text{Post}(P, \dot{\mathbf{x}})$, the time successor of P given a vector of variable rates $\dot{\mathbf{x}}$, by applying the two steps

1. Let each constraint $l_i \leq e_i \leq u_i$ where $\dot{e}_i \neq 0$ be written so that $\dot{e}_i > 0$, which can be achieved by multiplying the constraint by -1 if needed. For each distinct pair of constraints

$$l_i \leq e_i \leq u_i$$
$$l_j \leq e_j \leq u_j$$

   where $\dot{e}_i > 0$ and $\dot{e}_j > 0$, add to the set the constraint

$$\dot{e}_j l_i - \dot{e}_i u_j \leq \dot{e}_j e_i - \dot{e}_i e_j \leq \dot{e}_j u_i - \dot{e}_i l_j$$

2. Replace each constraint $l \leq e \leq u$ where $\dot{e} > 0$ by $l \leq e \leq \infty$.

We compute $\text{Free}(P, x)$, the result of unconstraining variable x in polyhedron P, using the two steps

1. Let each constraint $l \leq e \leq u$ in P where e has an instance of x be written in the form $l \leq cx - e' \leq u$, where e' involves the remaining variables and their coefficients and $c > 0$. For every distinct pair of such constraints in P

$$l_i \leq c_i x - e_i \leq u_i$$
$$l_j \leq c_j x - e_j \leq u_j$$

   combine the two in a way that cancels the x terms, adding to $\text{Free}(P, x)$ the constraint

$$c_j l_i - c_i u_j \leq c_i e_j - c_j e_i \leq c_j u_i - c_i l_j$$

2. Each constraint $l \leq e \leq u$ where e has no instances of variable x is added to $\text{Free}(P, x)$.

These algorithms might be viewed as generalizations of the difference methods used for timed automata[12, 8] and exhibit some similarity to the pragmatic algorithm used earlier for quantifier elimination[4]. Our prototype invokes a Simplex algorithm as part of the operations to test for feasibility and containment. We use a bounds tightening procedure to reduce the size of the constraint list after certain operations and to rapidly detect most infeasible polyhedra. Simplex-based reduction and feasibility testing is only applied when the bounds tightening procedure is ineffective. Details of our reachability analysis methods and implementation and proofs of correctness are documented elsewhere[29].

We benchmarked our prototype tool against HyTech and Verus[11] (a discrete timed automata reachability analysis tool that uses BDD techniques) using randomly generated uniprocessor workloads containing mixtures of periodic and aperiodic tasks. Figure 3 shows the percentage of problems that were solved by each of the tools, together with the primary reasons that solution was not achieved. Figure 4 compares the time required for solution for problems that were solved by all the tools using a logarithmic scale (a point appears for both HyTech and our prototype only for problems that were solved by both). We further increased the size of model we could analyze by applying some results from traditional scheduling theory to simplify the models, and by using a simple partial order reduction technique, these results are reported elsewhere[29].

5 Verifying the MetaH Executive

MetaH is an emerging SAE standard language for specifying real-time fault-tolerant high assurance software and hardware architectures[1, 24, 27]. Users specify how software and hardware components are combined to form an overall system architecture. This specification includes information about one or more configurations of tasks and message and event connections; and information about how these objects are mapped onto a specified hardware architecture. The specification includes information about timing behaviors and requirements, fault and error behaviors and requirements, and partitioning and safety behaviors and requirements.

Our current MetaH toolset, illustrated in Figure 5, can generate and analyze formal models for schedulability, reliability, and partition isolation. The toolset can also configure an application-specific executive to perform the specified task dispatching and scheduling, message and event passing, changes between alternative configurations, etc. Unlike many conventional

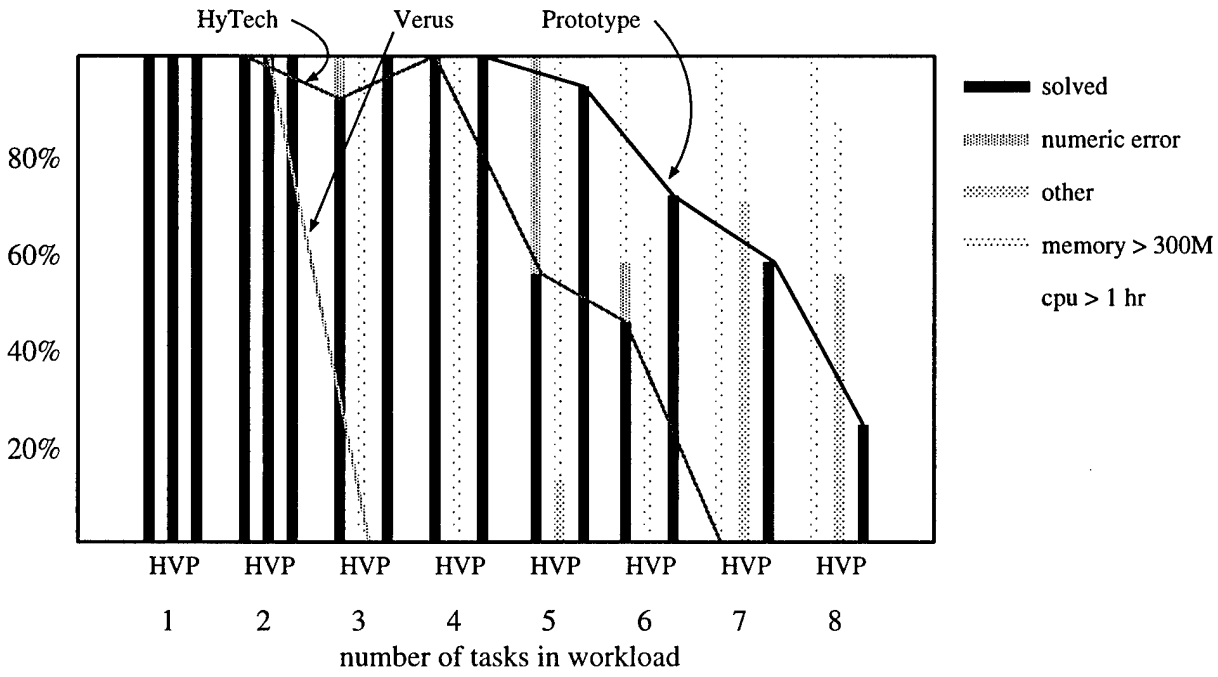


Figure 3: Percentage of Generated Problems That Were Solved

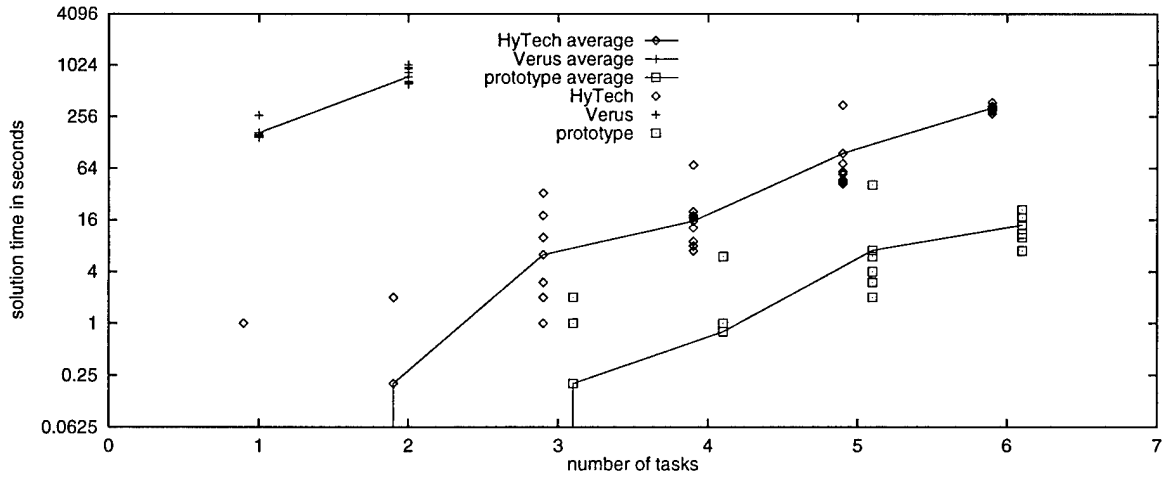


Figure 4: Solution Times for Problems That Were Solved

systems that rely on a large number of run-time service calls to configure a system by dynamically creating and linking to tasks, mailboxes, event channels, timers, etc., our toolset builds most of this information into an application-specific executive. There are relatively few run-time service calls, and the effects of these are tailored based on the specified application architecture and requirements.

Our MetaH executive supports a reasonably complex tasking model using preemptive fixed priority scheduling theory[5, 6, 7]. Among the features relevant to this study are period-enforced aperiodic tasks, real-time semaphores, mechanisms for tasks to initialize themselves and to recover from internal faults, and the ability to enforce execution time limits on all these features (time partitioning). Slack stealing in support of aperiodic and incremental tasks is also supported, but as we will mention later these were not modeled or verified.

Figure 6 shows the high-level structure of the

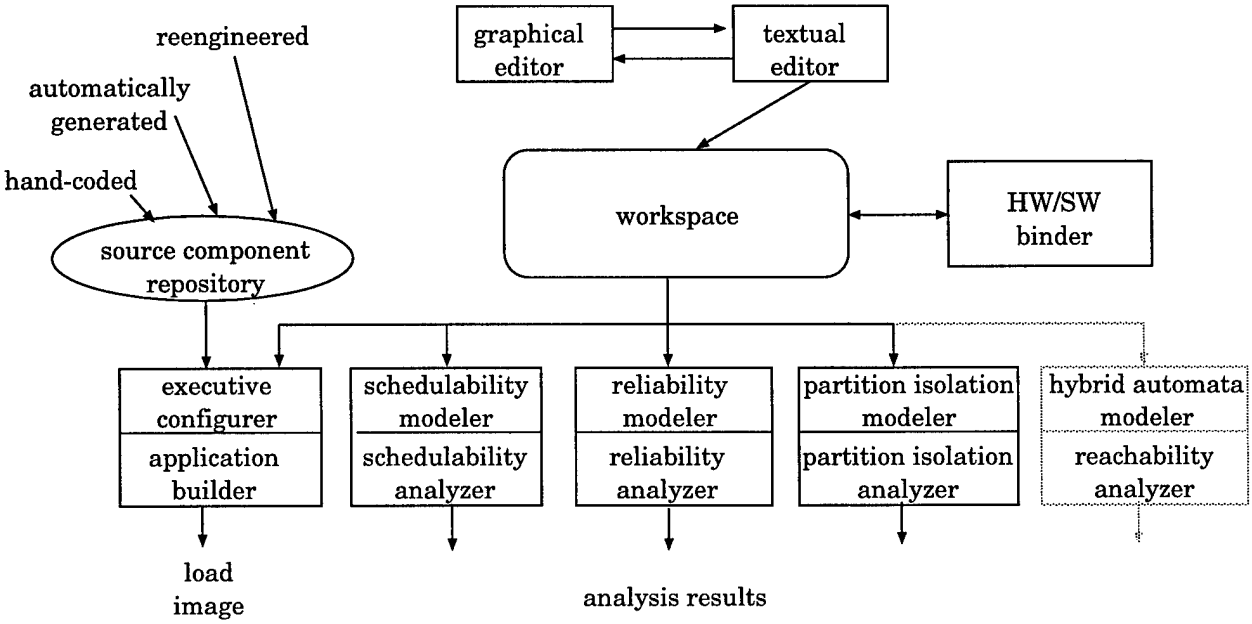


Figure 5: MetaH Toolset

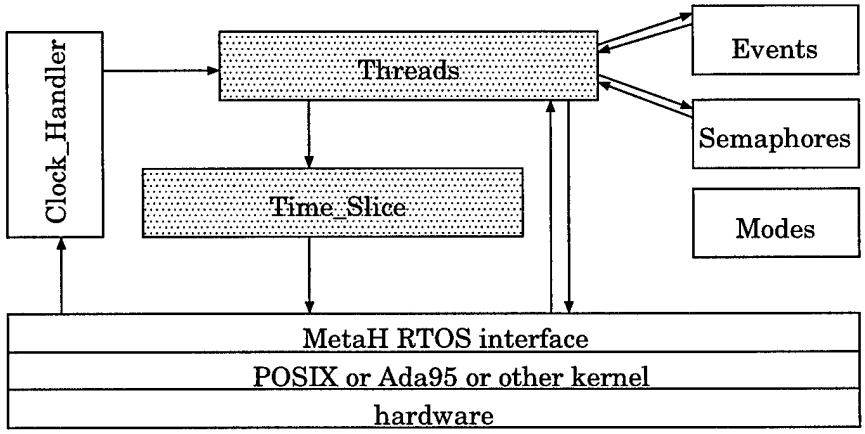


Figure 6: MetaH Executive Structure

MetaH executive. The core task scheduling operations are implemented by module `Threads`, e.g. start, dispatch, complete. These operations implement transitions between the discrete task scheduling states, e.g. dispatch may transition a task from the awaiting dispatch state to the computing state. These operations must take into account details such as the task type, optional execution time enforcement, event queueing, etc. Module `Threads` invokes operations in module `Time_Slice`, which encapsulates arithmetic operations and tests on two execution time accumulators maintained by the underlying RTOS and hardware for each task: an accumulator that increases while a task executes, and a time slice that decreases while a task executes. `Time_Slice` may set these variables to desired values using services provided through the `MetaH RTOS interface`. If time slicing is enabled for a task, then a trap will be raised by the underlying hardware and RTOS when the time slice reaches zero. This trap is handled by one of the operations in `Threads`. Module `Clock_Handler` is periodically invoked by the underlying system (it is the handler for a periodic clock interrupt) and makes calls to `Threads` to dispatch periodic tasks and start and stop threads at mode changes. Modules `Events`, `Modes` and `Semaphores` contain data tables and operations

to manage user-declared events, dynamic reconfiguration, and semaphores.

We produced hybrid automata models for the `Threads` and `Time_Slice` modules, about 1800 lines of code. We did not write a separate model using a special modeling language, instead we inserted calls to build the model into the executive code itself. For example, in the code that implements the dispatch operation there is logic to decide if a task can be dispatched, assignments to change program variables, and calls to set the time slice and execution time counters. Into this code we inserted a call to a modeling procedure to create an edge between the corresponding states of the linear hybrid automata model. The guards for this edge are the conditional expressions appearing in the code, and the assignments on this edge are the assignments appearing in the code. This provides a high degree of traceability between the implementation and the model.

The generation of the hybrid automata models resembled all-paths unit testing. We developed several simple application specifications that included most (but not all) of the tasking features. We wrote a test driver that exercised all relevant paths in the core scheduling modules. For each application specification, the test driver thus triggered the generation of a linear hybrid automata model of the possible behaviors of the core scheduling operations for a particular combination of tasks and features.

The conditions we checked during reachability analysis were that all deadlines were met whenever the schedulability analyzer said an application was schedulable; no accessed variables were unconstrained (undefined) and no invariants were violated on entry to a region; and no two tasks were ever in a semaphore locking state simultaneously. Assertion checks appearing in the code were modeled by edges annotated with `assert False`.

We also collected information about which edges were used by some transition during reachability analysis and compared this with all the possible edges that might be created (all instances of calls inserted into the code to create edges). This allowed us to insure that all modeled portions of the code were covered by at least one reachability analysis.

A total of 14 real-valued variables and 15 discrete states were defined to model each task. No single task model used all 14 variables and 15 states, different task types with different specified options used different combinations. Figure 7 shows the simplest linear hybrid automata model we generated, a periodic task with period and deadline of 100000us, compute time between 0 and 90000us, recovery time between 0 and 10000us. States are also annotated with processor scheduling priorities, which are not shown here. The variable rates were derived from the scheduling priorities by the analysis tool, which used preemptive fixed priority scheduling semantics for this study. Table 1 summarizes the complete set of applications we analyzed. A more detailed discussion of the modeling methods and results is provided elsewhere[30].

We discovered nine defects in the course of our verification exercise. Four of these were tool defects, two that could cause bad configuration data to be generated and two that could cause erroneously optimistic schedulability models to be generated. Six of these defects could cause errors only during the handling of application faults and recoveries, three of these six only in the presence of multiple near-coincident faults and recoveries. In our judgement, of the nine defects we found, one would almost certainly have been detected by moderately thorough requirements testing, while three would have been almost impossible to detect by testing due to the multiple carefully timed events required to produce erroneous behavior. The other five may have been detected by thorough requirements testing of fault and recovery features, providing the tester thought about possible execution timelines and arranged for tasks to consume carefully selected amounts of time between events.

There are a number of significant limitations on the degree of assurance provided. In our initial exercise, we chose not to model many behaviors that could have been modeled in a fairly straight-forward way, e.g. mode changes, inter-processor communication protocol, non-preemptable executive critical sections. In some cases different behaviors and subsystems can be modeled and analyzed almost independently, but it is not clear at what point the reachability analysis will become intractable as the extent of the model grows. Some behaviors might be more difficult to model, e.g. slack scheduling. The MetaH processor interface, underlying RTOS and hardware are unlikely to be fully model-able for a variety of practical and technical reasons. The MetaH tools were not verified, only a few specific generated modules and reports for a few example applications. Although our approach provides good traceability between code and model, there is still a very real possibility of modeling errors. The reachability analysis tool may contain defects; we discovered two in our tool in the course of this work. The modeled code does not change from application to application, and the analyzed applications fully exercised the code model, but to rigorously assert this

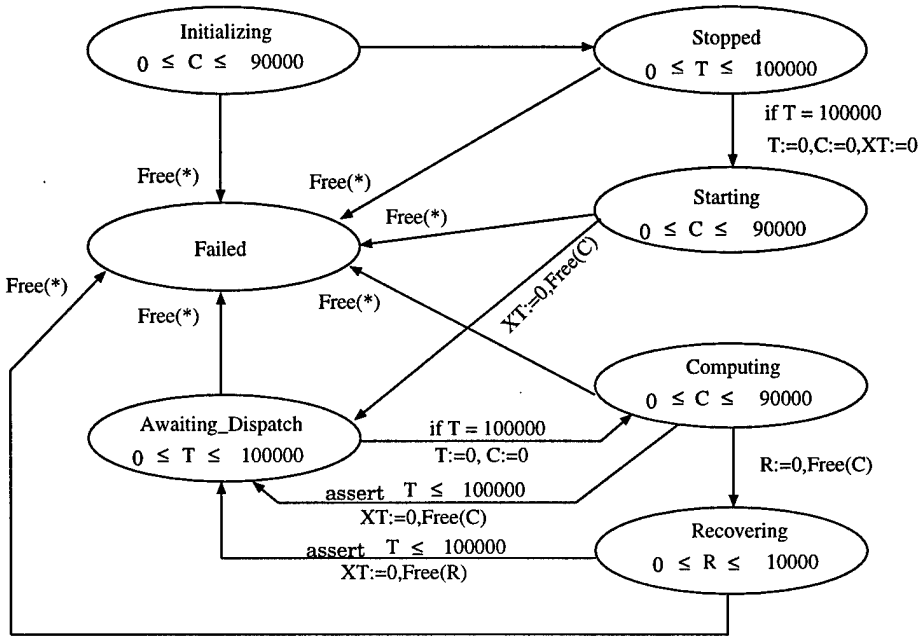


Figure 7: Generated Hybrid Automata Model for a Simple Periodic Task

Description	Discrete States	Distinct Polyhedra	Sparc Ultra-2 CPU Seconds
one periodic task	7	7	0
one periodic task, enforced execution time limits	7	10	0
one periodic task, enforced execution time limits, one semaphore	8	29	15
one period-enforced aperiodic task	9	18	0
one period-enforced aperiodic task, enforced execution time limits	9	27	2
one period-enforced aperiodic task, enforced execution time limits, one semaphore	11	124	125
two periodic tasks	36	60	3
two periodic tasks, enforced execution time limits	36	108	24
two periodic tasks, one with period transformed into two pieces,	41	97	10
two periodic tasks, one shared semaphore	48	118	36
two periodic tasks, one with period transformed into two pieces, enforced execution time limits	41	174	87
two periodic tasks, one with period transformed into four pieces, enforced execution time limits, recovery limit greater than compute limit	40	334	103
two tasks, one periodic and one period-enforced aperiodic	44	623	115
two periodic tasks, one with period transformed into four pieces, enforced execution time limits	41	351	170
two tasks, one periodic and one period-enforced aperiodic, enforced execution time limits	44	425	184
two tasks, one periodic and one period-enforced aperiodic, one shared semaphore	70	638	840
two periodic tasks, one with period transformed into two pieces, enforced execution time limits, one shared semaphore	55	963	5658

Table 1: Modeled Applications

code is correct for all possible applications would require some sort of induction argument. Even if the source code is correct, defects in the compiler, linker or loader software could introduce defects into the executable image.

Nevertheless, we estimate that the effort required for this exercise was roughly comparable to that required for traditional unit testing, but the results were more thorough than would have been achieved using traditional requirements testing. The method must be used in conjunction with traditional verification techniques such as testing, but it is at least intuitively reasonably easy to distinguish requirements that will be verified using hybrid automata from requirements that must be verified using other techniques.

6 Future Work

Our experience leads us to believe that linear hybrid automata are very powerful and well-suited for this domain. We were able to achieve one of our goals, the modeling and verification of a piece of real-world real-time software, with a number of limitations. We do not believe we have achieved the other goal yet, modeling and schedulability analysis for complex distributed systems of real-world size. However, there are a number of potential future developments that might reduce the verification limitations and provide useful schedulability analysis capabilities.

It should be possible to use the set of reachable regions produced by the analysis tool to automatically generate tests. This could significantly reduce the cost and increase the quality of requirements testing (which might still be required by the powers-that-be). Such tests could also detect defects that could not be found by model analysis, such as defects in the compiler, linker, loader, RTOS or hardware. One of the issues that must be confronted is the ease of constructing, running and observing the results of tests; for example, in theory one might encounter transitions in the model that occur only when two values are extremely close, which could be practically impossible to do in a test. Another issue is that such tests would not take into account the internal logic of unmodeled modules such as the RTOS; a systematic method for testing multiple points within each reachable polyhedron might help address this.

There are a number of potentially useful improvements in analysis methods and tools. Approximation and partial order methods might significantly increase the size of the model that could be analyzed[16, 19, 15, 29]. Preprocessing models to modify numeric parameters in certain ways can result in much more easily solved models[29]. It is possible to apply theorem proving methods to linear hybrid automata[21], and some work has been done on dense-time process algebras[10, 14]. Decomposition and induction methods currently being explored for discrete state models might be extensible to linear hybrid automata. There are a number of possible ways to visualize and navigate the reachable region space that would be of practical assistance during model development and debugging and during reviews. Concise APIs and support for inline modeling could reduce both the modeling effort and the number of modeling defects.

Changes will inevitably be required to the design, implementation and verification processes to make good use of these methods. Much of the benefit of other formal methods has been due to subsequent changes in development methods that resulted in more verifiable and defect-free specifications, designs and code in the first place. An important and not completely technical question is how verification processes might be changed to beneficially use these methods. What evidence would be required, for example, to convince a development organization or regulatory authority to replace selected existing verification activities with modeling and analysis activities, or to add modeling and analysis to current verification activities?

References

[1] *MetaH User's Guide*, Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN, `www.htc.honeywell.com/metah`.

[2] K. Altisen, G. GöBler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, "A Framework for Scheduler Synthesis," *Real-Time Systems Symposium*, December 1999.

[3] Rajeev Alur, Tomás Feder and Thomas A. Henzinger, "The Benefits of Relaxing Punctuality," *Proceedings of the Tenth Annual ACm Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 19-21, 1991.

[4] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.

[5] Pam Binns, "Scheduling Slack in MetaH," *Real-Time Systems Symposium*, work-in-progress session, December 1996.

[6] Pam Binns, "Incremental Rate Monotonic Scheduling for Im-

proved Control System Performance," *Real-Time Applications Symposium*, 1997.

[7] Pam Binns and Steve Vestal, "Message Passing in MetaH using Precedence-Constrained Multi-Criticality Preemptive Fixed Priority Scheduling," *submitted Real-Time Applications Symposium.*

[8] Johan Bengtsson and Fredrik Larsson, *UPPAAL, A Tool for Automatic Verification of Real-Time Systems*, DoCS 96/97, Department of Computer Science, Uppsala University, January 15, 1996.

[9] B. A. Brandin and W. M. Wonham, "Supervisory Control of Timed Discrete-Event Systems," *IEEE Transitions on Automatic Control*, v39, n2, February 1994.

[10] Patrice Brémond-Grégoire and Insup Lee, "A Process Algebra of Communicating Shared Resources with Dense Time and Priorities," University of Pennsylvania Department of Computer Science Technical Report MS-CIS-95-08, June 1996.

[11] S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, "Computing Quantitative Characteristics of Finite-State Real-Time Systems," *Real-Time Systems Symposium*, December 1994.

[12] David L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," *International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 12-14, 1989, also in *Lecture Notes in Computer Science 407*, J. Sifakis (Ed.), Springer-Verlag, pp 197-212.

[13] Andre N. Fredette and Rance Cleaveland, "RSTL: A Language for Real-Time Schedulability Analysis," *Proceedings of the Real-Time Systems Symposium*, December 1993.

[14] Andre N. Fredette, *A Generalized Approach to the Analysis of Real-Time Computer Systems*, Ph.D. Dissertation, North Carolina State University, March 1993.

[15] Nicolas Halbwachs, Pascal Raymond and Yann-Eric Proy, "Verification of Linear Hybrid Systems by Means of Convex Approximations," *Workshop on Verification and Control of Hybrid Systems*, Piscataway, NJ, October 1995.

[16] Nicolas Halbwachs, Yann-Erik Proy and Patrick Roumanoff, "Verification of Real-Time Systems using Linear Relation Analysis," *Formal Methods in System Design*, 11(2):157-185, August 1997.

[17] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri and Pravin Varaiya, "What's Decideable About Hybrid Automata?" *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995.

[18] Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, "HyTech: The Next Generation," *Real-Time Systems Symposium*, December 1995.

[19] Thomas A. Henzinger and Pei-Hsin Ho, "A Note On Abstract Interpretation Strategies for Hybrid Automata," *Hybrid Systems II*, also *Lecture Notes in Computer Science 999*, Springer-Verlag, 1995.

[20] Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, "A User Guide to HyTech," University of California at Berkeley, `www.eecs.berkeley.edu/~tah/HyTech`

[21] Thomas A. Henzinger and Vlad Rusu, "Reachability Verification for Hybrid Automata," *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control*, also *Lecture Notes in Computer 1386*, Springer-Verlag, 1998.

[22] Y. Kesten, A. Pnueli, J. Sifakis and S. Yovine, "Integration Graphs: A Class of Decideable Hybrid Systems," in R. L. Grossman, A. Nerode, A. P. Ravn and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, Springer-Verlag, 1993.

[23] Insup Lee, Patrice Brémond-Grégoire and Richard Gerber, "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," Department of Computer Science, University of Pennsylvania.

[24] Bruce Lewis, "Software Portability Gains Realized with MetaH, an Avionics Architecture Description Language," 18th *Digital Avionics Systems Conference*, St. Louis, MO, October 24-29, 1999.

[25] Anum Puri and Pravin Varaiya, "Decidability of Hybrid Systems with Rectangular Differential Inclusions," Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

[26] Peter J. G. Ramadge and W. Murray Wonham, "The Control of Discrete Event Systems," *Proceedings of the IEEE*, v77, n1, January 1989.

[27] Steve Vestal, "An Architectural Approach for Integrating Real-Time Systems," *Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1997.

[28] Steve Vestal, "Linear Hybrid Automata Models of Real-Time Scheduling and Allocation in Distributed Heterogeneous Systems," Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418, 1999.

[29] Steve Vestal, "A New Linear Hybrid Automata Reachability Procedure," Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418, 1999.

[30] Steve Vestal, "Formal Verification of the MetaH Executive Using Linear Hybrid Automata," Honeywell Technology Center, Minneapolis, MN 55418, December 1999.

[31] Jin Yang, Aloysius K. Mok and Farn Wang, "Symbolic Model Checking for Event-Driven Real-Time Systems," *ACM Transactions on Programming Languages and Systems*, v19, n2, March 1997.

Linear Hybrid Automata Modeling of Dynamic Real-Time Allocation and Scheduling in Distributed Heterogeneous Systems

Steve Vestal
vestal_steve@htc.honeywell.com
Honeywell Technology Center
Minneapolis, MN 55418*

Abstract

We propose that suitably extended linear hybrid automata provide a powerful way to model dynamic real-time allocation and scheduling in distributed heterogeneous systems, and that the associated theory can be used to address questions about the complexity of deciding various properties of such systems. In support of this we present a linear hybrid automata model of such systems and show that, with reasonable pragmatic restrictions, the schedulability problem is decidable.

1 Introduction

Most concurrent process models do not deal with resource allocation and scheduling[21, 15], while most real-time preemptive scheduling models do not deal with complex process sequencing and interaction[18, 6]. This paper presents a dense time hybrid automata model that allows systems of concurrent processes to be dynamically allocated and preemptively scheduled for real-time execution on a heterogeneous set of resources. The model is powerful enough to allow a variety of scheduling disciplines, including nonpreemptive, preemptive fixed priority, and earliest deadline. The model is powerful enough to allow a variety of dynamic reallocation algorithms. The model allows compute times to be specified using intervals, which makes them suitable for use in multiprocessor systems that exhibit anomalous scheduling behavior[12]. We allow timer rates to vary nondeterministically within specified ranges, so that clock drift and uncertainty can be modeled. Tasks, their internal behaviors, and their external interactions can be modeled as concurrent finite state machines, which allows modeling of remote procedure calls, rendezvous, state-dependent variations in compute time, and implementation details of code for synchronization and communication protocols.

We use a linear hybrid automata model, extended in certain ways to more directly model resource allocation and scheduling of real-time tasks in distributed heterogeneous systems. We use a concurrent finite state automaton model of the processes, to which we add a set of continuous variables to model time and accumulated work, a set of resources to service the activities comprising each process, and a function that specifies how activities are allocated to and scheduled on each resource. We allow allocation and scheduling choices to dynamically change at any discrete system event. The schedulability problem for a real-time system can be reduced to the problem of determining whether a discrete system state is reachable in a model.

Accumulated execution time is modeled using continuous variables that can be stopped and started, which is known to make the reachability problem undecidable for linear hybrid automata. However, the classical construction used to prove undecidability includes features that would be considered defects in the design of an actual system. We prove that the reachability problem is decidable given various restrictions on the use of equality comparisons in transition guards, where these restrictions are reasonable in practice for most systems. Our proof makes use of a discrete time finite automaton construction and may be of some interest in relating continuous time and discrete time models.

The construction used in our proof does not yield a practical algorithm for testing discrete state reachability and hence schedulability. Rather, the purpose of this paper is to provide some work that illustrates two theses. First, appropriately extended linear hybrid

*This work has been supported by the Air Force Office of Scientific Research under contract F49620-97-C-0008.

automata models provide a powerful way to describe many aspects of systems that are beyond the reach of current theory. Second, linear hybrid automata models may be useful in exploring the computational complexity of complex resource allocation and scheduling problems. For example, our work leaves open questions about the decidability of systems in which tasks are dynamically reallocated between processors of different rates in mid-execution, and the decidability of systems whose scheduling decisions are based in part on comparisons between accumulated compute times.

There is a substantial body of work on the use of discrete timed automata models for real-time systems[8, 10, 17, 20, 19, 22]. Our work is distinguished from this by our use of a dense time model. Some work exists on dense time process algebras[11, 7], we use an automata model. There is also a large body of work on timed and linear hybrid automata[2, 3, 1, 4, 13]. We extend these models to include resource allocation and scheduling and focus on the modeling and analysis of these aspects of real-time distributed heterogeneous systems.

2 Real-Time Concurrent Automata

We use real numbers $\Re$ to model time, but we assume that values appearing in model specifications or manipulated by computer are rationals. We make use of intervals of rational numbers, elements of which are written as $[a, b], a \leq b$. We will overline interval variables to distinguish them from rational variables, e.g. x is a real-valued or rational-valued variable while $\bar{x}$ denotes some rational interval $[a, b]$.

We use several sets of objects as variables whose values change at discrete instants of time. For a set of variables V whose values range over R we use a set of assignment functions $\{\alpha | \alpha : V \rightarrow R\}$, where each α specifies a particular association of values with objects. An assignment operation $v \leftarrow r$ at some time t changes the assignment function in effect. We sometimes use subscripts to distinguish assignment functions before and after assignment operations, e.g. α_i becomes α_{i+1} after assignment $v \leftarrow r$ where $\alpha_{i+1}(v) = r$.

An important class of assignments are those for objects whose range $R = (\Re^+ \rightarrow \Re^+)$ is a set of functions that map nonnegative real time to nonnegative scalar real values. In such cases $\alpha(v)$ is a function and $\alpha(v)(t)$ is the value of that function at time t. The assignment operation $v \leftarrow f$ changes the time-varying function associated with variable v.

The basic elements of our real-time concurrent automata model are a finite set of process variables that range over finite sets of activities, a finite set of integrators that range over real-valued functions of time, a finite set of resources to service process activities, and a finite set of edges that determine the possible discrete changes in system state.

- $P = \{p_1, p_2, \cdots, p_m\}$ is a finite set of discrete state variables that we will call processes. Each process p_i ranges over a finite set of states or locations $A_i = \{a_{i1}, a_{i2}, \cdots, a_{in_i}\}$ that we will call activities. If the boolean predicate $\text{init}(a_{ij})$ is true then a_{ij} is said to be an initial activity of p_i. We require there be at least one initial activity in each A_i. Distinct activity sets are disjoint, $A_i \bigcap A_j = \emptyset, i \neq j$. The set of all activities in a system is $A = \bigcup_i A_i$, and the discrete state space of a system is $A^{\otimes} = A_1 \times A_2 \times \cdots \times A_m$. We use $\{\alpha \mid \alpha : P \rightarrow A, \alpha(p_i) \in A_i\}$ as the set of functions that associate each process with a current activity, and we sometimes write $a_{ij} \in \alpha$ to mean $\alpha(p_i) = a_{ij}$. When $\alpha(p_i) = a_{ij}$ changes to $\alpha'(p_i) = a_{ik}$ we say that activity a_{ij} completes and activity a_{ik} is dispatched.

- $X = (X_t = \{x_1, x_2, \cdots, x_n\}) \bigcup A$ is a finite set of function-valued state variables that we will call integrators. We use $\mathcal{F} = \{\nu \mid \nu : X \rightarrow (\Re^+ \rightarrow \Re^+)\}$ to associate a real-valued function of time with each integrator. The integrators $x_1, x_2, \cdots, x_n$ will be used as resetable clocks or timers, and integrators a_{ij} will be used to record accumulated work performed on activity a_{ij} (i.e. activities in A are associated with time-varying functions by assignments in $\mathcal{F}$).

  Rather than directly specify an assignment $\nu(x)$ for an integrator x, we will instead specify the value at some point $\nu(x)(t)$ together with constraints on the derivative $\dot{\nu}(x)$. For functions that satisfy the Lipshitz conditions, the value at a point plus the derivitive is sufficient to uniquely determine the function (i.e. $\nu(x) = \int \dot{\nu}(x)$ plus a suitable constant).

- $\Sigma = \{l_1, l_2, \cdots, l_k\}$ is a finite set of labels or letters. Elements of Σ will be used to label activity dispatches and completions in the various processes. It is sometimes helpful to think of Σ as containing disjoint subsets Σ_{p_i} of letters that are associated exclusively with individual processes p_i plus additional letters used to label joint or synchronized dispatches and completions involving multiple processes.

- $R = \{r_1, r_2, \cdots, r_n\}$ is a finite set of resources. The partial assignments $\Pi = \{\pi \mid \pi : (A \bigcup E) \rightarrow$

$(R \bigcup \{\phi\})\}$ specify whether an activity is executing and, if so, on which resource. If $\pi(a)=r$ then activity a is executing on resource r, if $\pi(a)=\phi$ then a is not executing. The range of assignments in Π also includes a set E of edges that we will discuss shortly, where $\pi(e)\neq\phi$ means that an edge is enabled. Note that the form of Π restricts an activity to be executing on at most one resource, allows a resource to be concurrently executing more than one activity, and allows multiple edges to be concurrently enabled.

The allocation and scheduling function $s : 2^E \times A^{\otimes} \rightarrow \Pi$ maps a set of edges and a set of current activities to an element of Π. The allocation and scheduling function, which is a parameter of the model, will be evaluated at each activity completion to determine a new value for π. We use $s(\alpha_0)$ to denote the initial allocation and scheduling decision for a given choice of initial activities α_0.

The function $\bar{\omega}(r_i, a_{jk}) \in \mathcal{I}^+$ specifies an interval containing the rate or speed at which resource r_i can service activity a_{jk}. When $\pi(a) = r$ then a is accumulating service from resource r at some rate in the interval $\bar{\omega}(r, a)$. We overload $\bar{\omega}$ and apply it to timers as well, where $\bar{\omega}(x) \in \mathcal{I}^+$ is some interval of possible rates for timer x. We sometimes abuse notation and write simply $\bar{\omega}(x)$ for x an activity as well as a timer when the resource r_i is implicitly bound or quantified. $\bar{\omega}$ is fixed in a model and does not change over time.

- $E : A^{\otimes} \times \mathcal{F} \times \Pi \rightarrow A^{\otimes} \times \mathcal{F} \times \Pi$ is a mapping between system state values that defines the possible discrete transitions of a system. We will define this mapping using a finite subset

$$E \subseteq A \times 2^{\{(x,\bar{c}) \mid x \in X, \bar{c} \in \mathcal{I}^+\}} \times \Sigma \times A \times 2^X$$

whose elements we will call edges.

An edge

$$e = \langle a_{ij}, \mathcal{C} = \{(x, \bar{c}) \mid x \in X, \bar{c} \in \mathcal{I}^+\}, l, a_{ik}, \mathcal{R} \subseteq X \rangle$$

means that at time t, a system in which process p_i is executing activity a_{ij} and $\nu(x)(t) \in \bar{c}$ for a specified subset of the integrators might complete activity a_{ij}, dispatch activity a_{ik}, and change $\nu(x)$ so $\nu(x)(0) = 0$ for a specified subset of the integrators. This discrete event is labeled l. We use e.src, $e.\mathcal{C}$, $e.l$, e.dst and $e.\mathcal{R}$ to conveniently refer to specific elements of edge e.

We use $E(\alpha) = \{e \mid e.\text{src} \in \alpha\}$ to denote the set of edges whose source activities are current in the discrete system state α. We use $E_l(\alpha) = \{e \in E(\alpha) \mid e.l = l\}$ to denote the set of edges labeled with $l \in \Sigma$ whose source activities are current. Similarly, we use $\mathcal{C}_l(\alpha) = \bigcup e.\mathcal{C}$ to be the union of all the constraints $(x, \bar{c})$ for all edges $e \in E_l(\alpha)$, and $\mathcal{R}_l(\alpha) = \bigcup e.\mathcal{R}$ to be the set of all the integrators to be reset for all edges $e \in E_l(\alpha)$. We sometimes refer to a set $E_l(\alpha)$ as a set of coedges.

Note that edges do not allow a direct specification of changes to $\dot{\nu}$ at transition events. In our model $\dot{\nu}$ will be determined by π and $\bar{\omega}$.

A discrete system state of a real-time concurrent automata is defined by specifying current activities for all processes, functions for all integrators, and allocation and scheduling choices for activities and edges,

$$\mathcal{S} = \langle\langle \alpha, \nu, \pi \rangle\rangle$$

We sometimes use $\mathcal{S}.\alpha$, $\mathcal{S}.\nu$ and $\mathcal{S}.\pi$ to denote particular elements of a system state $\mathcal{S}$.

A trace or run of a system is a sequence of states

$$\mathcal{S}_0 \xrightarrow{t_1} \mathcal{S}_1 \xrightarrow{t_2} \mathcal{S}_2 \xrightarrow{t_3} \cdots$$

where $\mathcal{S}_i$ is the state between times t_i and t_{i+1} at which discrete transition events occur. $\mathcal{S}_0$ at $t_0 = 0$ is the initial system state. Each discrete system state represents a possibly uncountably infinite number of continuous states and can be thought of as a continuous function of time with value $\mathcal{S}_i(t) = \langle\langle \alpha_i, \nu_i(t), \pi_i \rangle\rangle$ at time $t_i \leq t \leq t_{i+1}$.

What state is the system in at exactly t_i? An intuitively appealing approach is to say that adjacent intervals have appropriately matched open and closed ends, e.g. $[0, t_1)$, $[t_1, t_2]$, $(t_2, t_3), \cdots$. Such traces are called strongly monotonic[4]. However, we need to deal with sequences of simultaneous transitions that all happen at the same instant of time. We therefore adopt a weakly monotonic trace semantics in which all these intervals are closed-ended. This means a system is in multiple, temporally indistinguishable states at transition instants: at time t_i there may be two value assignments for an integrator, two current activities for a single process, etc. The mapping from true time to system state is thus not a 1-to-1 function even along a single execution trace. The value of integrator x in state $\mathcal{S}_i$ at time t_i is $\nu_i(x)(t_i)$ and at time t_{i+1} is $\nu_i(x)(t_{i+1})$, the latter also being the value of $\nu_{i+1}(x)(t_{i+1})$ when x is not reset at time t_{i+1}.

We can now define system dynamics by specifying the set of all possible traces of a given real-time concurrent automaton.

$\mathcal{S}_0$, the initial state of a possible trace, is any state $\langle\langle \alpha_0, \nu_0, \pi_0 \rangle\rangle$ where

$$\forall p \in P, \text{init}(\alpha_0(p))$$
$$\pi_0 = s(\alpha_0)$$
$$\forall x \in X, \nu_0(x)(0) = 0$$
$$\forall x \in X, t \in \Re^+, \begin{cases} \dot{\nu}_0(x_i)(t) \in \bar{\omega}(x_i) & \text{if } x_i \in X_t \\ \dot{\nu}_0(x)(t) \in \bar{\omega}(\pi_0(x), x) & \text{if } x \in \alpha_0 \\ & \quad \bigwedge \pi_0(x) \neq \phi \\ \dot{\nu}_0(x)(t) = 0 & \text{otherwise} \end{cases}$$

The function $\nu_0(x)$ is specified by an initial condition at time t_0 together with a derivative function $\dot{\nu}(x)$. The integrator derivative $\dot{\nu}(x)$ is a continuous integrable function whose value stays within the specified interval. The service rate $\dot{\nu}(x)$ for an executing activity may vary within an interval that depends on the activity and the resource (i.e. on π). The service rate for non-executing activities is 0.

A trace $\mathcal{S}_0 \xrightarrow{t_1} \cdots \xrightarrow{t_i} \mathcal{S}_i$ can progress to a new system state $\mathcal{S}_{i+1}$ by a transition labeled l over coedges $E_l(\alpha_i)$ at time t_{i+1} if $E_l(\alpha_i) \neq \emptyset$ and the following conditions hold.

- $\forall e \in E_l(\alpha_i), \pi_i(e) \neq \phi$
- $\forall e \in E_l(\alpha_i), \pi_i(e.\text{src}) \neq \phi$
- $\forall (x, \bar{c}) \in C_l(\alpha_i), \nu(x)(t_{i+1}) \in \bar{c}$

The next state $\mathcal{S}_{i+1}$ is

$$\forall p \in P, \alpha_{i+1}(p) = \begin{cases} e.\text{dst} & \text{for } e.\text{src} = \alpha_i(p), e \in E_l(\alpha_i) \\ \alpha(p) & \text{otherwise} \end{cases}$$

$$\pi_{i+1} = s(E_l(\alpha_i), \alpha_i)$$

$$\forall x \in X, \nu_{i+1}(x)(0) = \begin{cases} 0 & \text{if } x \in \mathcal{R}_l(\alpha_i) \\ \nu_i(x)(t_{i+1}) & \text{otherwise} \end{cases}$$

$$\forall x \in X, t \in \Re^+, \begin{cases} \dot{\nu}_{i+1}(x)(t) \in \bar{\omega}(x_i) & \text{if } x_i \in X_t \\ \dot{\nu}_{i+1}(x)(t) \in \bar{\omega}(\pi_{i+1}(x), x) & \text{if } x \in \alpha_{i+1} \\ & \quad \bigwedge \pi_{i+1}(x) \neq \phi \\ \dot{\nu}_{i+1}(x)(t) = 0 & \text{otherwise} \end{cases}$$

Each time-varying function $\nu_{i+1}(x)$ is uniquely determined by the value $\nu_{i+1}(x)(t_{i+1})$ and the choice of derivative function $\dot{\nu}_{i+1}(x)$. The nondeterminism is in the choice of transition time t_{i+1}, choice of transition edge(s), and choice of derivative functions. Note that the only values of $\pi(a)$ used to determine $\dot{\nu}$ are the values for current activities $a \in \alpha$, values for non-current activities are ignored.

Let $\nu(x)([t_i, t_j])$ denote the function $\nu(x)$ in the interval $[t_i, t_j]$. If $\nu_0(x), \nu_1(x), \cdots$ is a sequence of ν assignments for integrator x with state transitions occurring at times $t_1, t_2, \cdots$ then we call the function given by the union of $\nu_0(x)([t_0, t_1]), \nu_1(x)([t_1, t_2]), \cdots$ the trajectory of integrator x. We use $x(t)$ to denote the trajectory of x. The trajectory $x(t)$ is what is usually thought of as the value of x over time as the system operates.

3 Reachability

Contemporary models of hybrid automata associate with each discrete state bounds on the values of continuous variables. These are used to force progress, to prevent a system from staying in a discrete state without ever taking a transition. Reachability can also be defined in terms of the total state space, i.e. whether both a discrete state and a set of variable values satisfying some equation can be reached.

For our purposes, however, we can restrict ourselves to considering the problem of whether a discrete state is reachable or not. What we can do is add discrete states that represent violations of a desired assertion, where the edges to those states have guards constructed so that the discrete state is reachable only when the assertion fails. For example, we can add to each process a state representing a missed deadline. From each activity we add an edge to this state, where the guard on this edge is that the deadline has elapsed but the maximum compute time has not been accumulated. This discrete state is reachable only if these conditions can occur in the system.

4 Modeling Real-Time Systems

In this section we outline how our model can be used to describe actual systems. We focus on modeling computer system processes and resources rather than environment processes, although the formalism is also intended to model environment timing. We begin by providing an example of a simple periodic process executing on a uniprocessor, illustrated graphically in Figure 1.

We model a processor using a pair of resources, r_x to service the executing activity and r_w to service activities that are waiting for events (think of waiting as a kind of activity). $\pi(a) = r_x$ for at most one a, the activity selected for execution by the scheduling policy. $\pi(a) = r_w$ for all waiting activities, which are all serviced concurrently without contention. $\pi(a) = \phi$ for compute activities that have been dispatched but not selected for execution by the scheduling policy. As mentioned earlier, $\pi(a)$ is only relevant for a a current activity, and we adopt the convention that $\pi(a) = \phi$

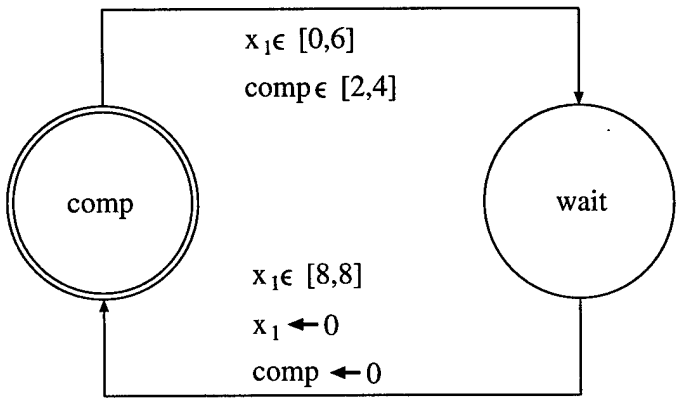


Figure 1: Periodic Process

if a is not a current activity.

We model a periodic process as a pair of alternating activities `comp` and `wait`, where `comp` is the initial activity. `Comp` is also used as an integrator that records the amount of work performed on activity `comp`. $\nu(\texttt{comp})$ is initially zero and is reset at each new dispatch of activity `comp`, so $\nu(\texttt{comp})(t)$ is the work performed up to time t since the most recent dispatch. The edge constraint $\texttt{comp} \in [2,4]$ in figure 1 means that activity `comp` cannot complete until the accumulated work falls somewhere between 2 and 4, the actual execution time being otherwise variable or uncertain. We also use a timer x_1 in figure 1 to model deadlines and periodic dispatching. The constraint $x_1 \in [0,6]$ imposes a preperiod deadline of 6 on the computation, the constraint $x_1 \in [8,8]$ causes `comp` to be dispatched 8 units of time after the previous dispatch (x_1 is initially 0 at the first dispatch and is reset to 0 at each subsequent dispatch).

Figure 2 shows one possible sequence of π and ν assignments for two periodic processes scheduled preemptively with a deadline monotonic fixed priority assignment. For current compute activities (those that have been dispatched but not completed), $\pi(a) = r_x$ for the highest priority activity a and $\pi(b) = \phi$ for all lower-priority activities b. For all current wait activities $\pi(a) = r_w$. The vertical lines show where transition events occur. Timer and processor rates are assumed close to 1 in this graph, so accumulated work increases at about the same rate as elapsed real time when an activity is executing (the functions for timers are not shown in figure 2, they look like ramp functions with periods of 8 and 16).

Figures 1 and 2 impose no constraints on, and never reset, $\nu(\texttt{wait})$. We assume π executes all current waiting activities on r_w, and $\nu(\texttt{wait})$ can be used to record accumulated wait time for these activities. Constraints on these values can be used to impose limits on wait time when desired.

We use non-singular interval constraints on accumulated compute time to model uncertainty or variability in execution times. An activity completes after some varying and/or uncertain number of processor cycles that corresponds to some execution time value in the specified interval. When the activity completes the process transitions to a new activity. A rendezvous between two computing activities occurs only when they complete at exactly the same instant, which in practice would require that one activity busy-wait for the other. A common implementation, in which the first activity to reach a rendezvous point waits until a suitable partner is ready, could be modeled by introducing a waiting activity as the source for edges with global labels. A specific rendezvous semantics will in general have associated modeling guidelines and restrictions, and the scheduling function s may need to enable and disable edges appropriately.

The model is powerful enough to admit many common scheduling disciplines, including preemptive fixed priority and earliest deadline. However, schedulers that make decisions based in part on accumulated or actual compute times, such as least laxity and slack scheduling, cannot be modeled since ν is not a parameter of the scheduling function s (we shall motivate this exclusion later in the paper).

A single contention-free wait resource can be shared in multiprocessor models, so that n processors and busses can be modeled with $n+1$ resources (except when wait resources having distinct rates are needed). The modeling of heterogeneous systems is supported by the ability to specify different service rates for different activities and resources. Using intervals rather

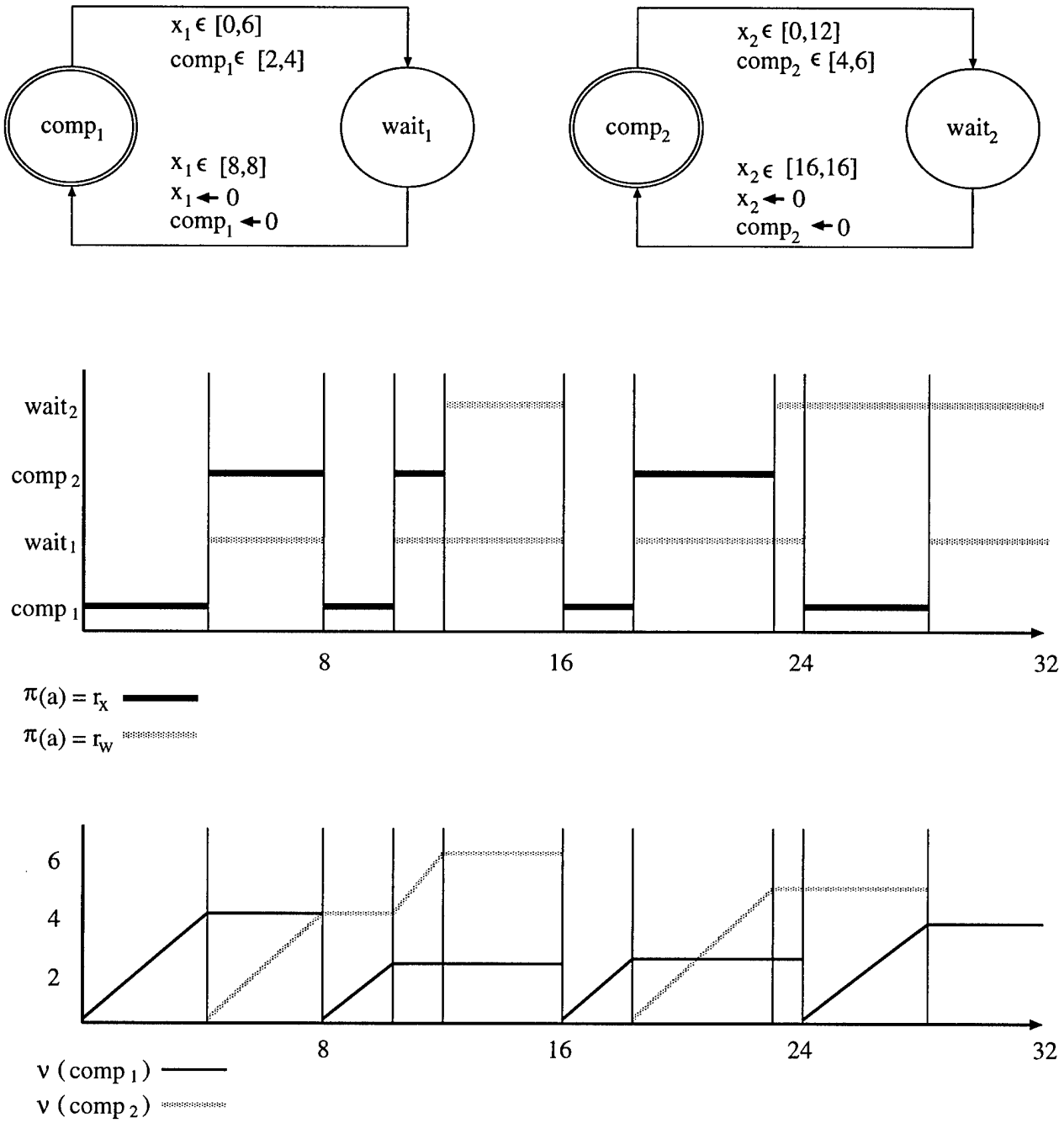


Figure 2: Values for π and ν using Deadline Monotonic Scheduling

than scalars allows timer drift to be modeled in distributed systems. Timers and processors are distinct objects in the modeling formalism, e.g. there is no concept of a timer being associated with or accessible only from a specific processor. Any such assumptions or restrictions need to be reflected in specific models.

5 Decidability

A recurring result of hybrid automata theory is the undecidability of the discrete state reachability problem for reasonably general models. We now show that the reachability problem for our model is undecidable. We then give pragmatically reasonable restrictions that make the problem decidable. Our decidabil-

ity proof is based on the construction of a finite-state discrete time machine whose reachability is equivalent to that of a given continuous time system, and so as a side-effect we show that our decidable continuous time model is fundamentally no more powerful than a (combinatorially large) finite-state discrete time model. Before giving this proof, however, we first outline a proof of the known undecidability result. This provides some insight into the restrictions we impose to achieve decidability.

Theorem 1 *Given arbitrary $\alpha' \in A^{\otimes}$ it is undecidable if there exists a trace containing a state $\mathcal{S}$ where $\mathcal{S}.\alpha = \alpha'$ (it is undecidable if the discrete system state α' is reachable).*

Proof 1 The discrete state reachability problem for hybrid automata is known to be undecidable if there is a choice of at least two rates for different integrators, or if one integrator can be stopped and later continued[16, 13]. Our model allows both. The proof is by reduction to the reachability problem for two-counter machines.

A two-counter machine is a finite-state control augmented with two non-negative integer counters. A transition in the finite-state control may be conditioned on either counter being zero, and a transition may increment or (for non-zero counters) decrement either counter. A two-counter machine is equivalent in power to a Turing machine, and the reachability problem for two-counter machines is undecidable.

The essence of the proof of the undecidability of the discrete state reachability problems for continuous time systems is to construct, for a given two-counter machine, a continuous time system that encodes counter value c as an integrator value $\frac{1}{2^{c+1}}$. Increments and decrements of counters in the original two-counter machine are implemented by halving and doubling integrators in the continuous time system. Any solution of the discrete state reachability problem for continuous time systems could thus be used to solve the reachability problem for two-counter machines, which would contradict the undecidability of the latter problem.

The state transition diagram of the two-counter machine is modified by adding continuous variables for each of the counters, a clock variable that is used to synchronously drive transitions in the constructed hybrid automaton, plus a temporary variable. The proof relies on constructions for assigning the value of one variable to another, for doubling the value of a variable, and for halving the value of a variable.

Figure 3 illustrates the construction for doubling a variable by showing the time line for the values of

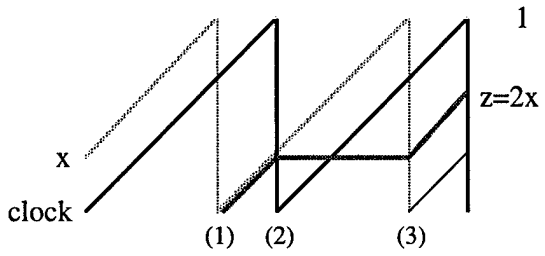


Figure 3: Construction for Doubling a Value

three variables. The guard for every edge in the constructed hybrid automaton has a clause that restricts transitions to occur only when the clock equals 1, and the clock is reset at each transition. Edges are added to all states so that whenever a counter variable equals 1 then it is reset. The counter value encoded into a variable is the log of the inverse of the value of that variable at each clock reset, minus one. The value can be viewed as encoded in the phase shift between the clock and the counter variable, where each counter variable is reset x time units before the clock reset, $x = \frac{1}{x^{c+1}}$.

To double a value, a doubling variable z is reset to 0 at the same time as the variable x whose value is to be doubled. This point is marked (1) in the figure. At the next clock reset, the rate for this variable is set to 0, marked (2), causing it to store the encoded counter value. The rate is set back to 1 when the counter variable is reset, marked (3). This occurs x time units before the clock reset, so at the next clock reset the doubling variable has a value of $2x$.

A variable value is halved by setting a halving variable to 0 at some nondeterministic time, then doubling that value, then having an edge and guard that allow a transition only if the doubled value equals the value being halved (which leaves the original nondeterministically set halving variable equal to the desired value.).

The above construction is possible with a fairly simple continuous time system model, one with two processes on a single processor and all integrator rates equal to 1 (except suspended activities with integrator rates of 0). However, it would be difficult to implement such a construction in practice because it is difficult to trigger an event only if and when two timer equality tests, or equality tests for a timer and an accumulated execution time, succeed at the same instant (as is done to halve an integrator to simulate a decrement). In fact such conditions are usually avoided in the design of actual real-time systems. The ability to test for equality is known to add theoretical power and

complexity to continuous time temporal logics[5]. We next prove that the reachability problem becomes decidable if we place pragmatic restrictions on the ability to do exact equality tests.

Theorem 2 *The discrete state reachability problem is decidable for a continuous time system if the following conditions hold.*

- *The set of possible outputs of the scheduling function s for each possible set of input values is finite and can be enumerated.*
- *For every integrator $x \in A$ the rate interval $\bar{\omega}(x)$ is fixed between resets of that integrator (i.e. the scheduler does not dynamically reallocate any activity in mid-execution across processors having different rates for that activity).*
- *For every integrator x, either every edge constraint $(x, \bar{c})$ for x has a non-singular interval $\bar{c}$, or else the rate interval $\bar{\omega}(x)$ is non-singular (i.e. system behavior does not depend on exact equality comparisons with exact drift-free timers or execution rates).*
- *However, we allow as a special exception integrators whose rate interval and some of whose edge constraints are singular, providing the integrator is only reset or stopped or started at a transition having at least one edge constraint $(y, \bar{d})$ with $\bar{d}$ and $\bar{\omega}(y)$ singular (y may but need not be x), and for every such singular constraint on that edge $\bar{\omega}(x) = k\bar{\omega}(y)$ for some positive integer k (i.e. some types of noninteracting or harmonically interacting behaviors may be modeled exactly).*

Proof 2 We construct a finite-state discrete time system in which a discrete system state is reachable iff that same discrete system state is reachable in a given continuous time system. The reachability problem for the continuous time system can then be answered by exhaustively searching the reachable state space of the finite-state system. The assumptions of the theorem make it possible to approximate any continuous trajectory $x(t)$ with a piece-wise linear trajectory that satisfies the same set of rate and edge constraints and enables the same set of transitions. We first present the construction and then prove the equivalence of the continuous time and discrete time reachable discrete state spaces.

The initial states $\mathcal{S}'_0$ of the discrete time model at time 0 are identical to those of the continuous time model,

$$\begin{aligned} &\forall p \in P, \mathrm{init}(\alpha_0(p)) \\ &\forall x \in X, \nu_0(x)(0) = 0 \\ &\pi_0 = s(\alpha_0) \end{aligned}$$

Let $\{c_1, c_2, \cdots\}$ be the set of all non-zero non-infinite end-points of edge constraint intervals that appear in a model, let $\{\omega_1, \omega_2, \cdots\}$ be the set of all non-zero non-infinite end-points of rate intervals that appear in a model, and let $\{\frac{c_i}{\omega_j}, \cdots\}$ be the set of ratios where there is some integrator x that can have minimum or maximum non-zero non-infinite rate ω_j and is compared on some edge with an interval having non-zero non-infinite end-point c_i. It is known that for any finite set of rational numbers there exists a unique greatest common divisor, itself a rational number, where every element of the set is an integer multiple of this greatest common divisor. Let Δ be the greatest common divisor of the set $\{c_1, c_2, \cdots, \omega_1, \omega_2, \cdots, \frac{c_i}{\omega_j}, \cdots\}$, and let $\mathbf{M}$ be the largest of $\{c_1, c_2, \cdots\}$. We use $\lfloor q \rfloor_\Delta$ ($\lceil q \rceil_\Delta$) to denote the largest integer multiple of Δ less than or equal to (smallest integer multiple of Δ greater than or equal to) some rational number q.

A discrete time system progresses from system state $\mathcal{S}'_i$ to a new system state $\mathcal{S}'_{i+1}$ by a transition along coedges $E_l(\alpha'_i)$ if the same conditions given earlier for edge transitions in the continuous time system hold. The next discrete time state $\mathcal{S}'_{i+1}$ is

$$\forall p \in P, \alpha_{i+1}(p) = \begin{cases} e.\mathrm{dst} & \text{for } e.\mathrm{src} = \alpha_i(p), e \in E_l(\alpha_i) \\ \alpha(p) & \text{otherwise} \end{cases}$$

$$\forall x \in X, \nu_{i+1}(x) = \begin{cases} 0 & \text{if } x \in \mathcal{R}_l(\alpha_i) \\ \nu_i(x) & \text{otherwise} \end{cases}$$

$$\pi_{i+1} = s(E_l(\alpha_i), \alpha_i)$$

If the scheduling function s is nondeterministic then there is actually a set of successor states, one for each possible scheduling decision. By assumption this set is finite and enumerable.

In addition to transitions over edges, the discrete time system can progress by allowing another Δ unit

of time to pass,

$$\forall x \in X, \nu_{i+1}(x) = \begin{cases} \nu_i(x) + n\Delta & \text{if } \nu_i(x) \le \mathbf{M}, \\ & x_i \in X_t, \\ & n\Delta \in \bar{\omega}(x_i) \\ & \quad \text{for } n \in \mathcal{Z}^+ \\ \nu_i(x) + n\Delta & \text{if } \nu_i(x) \le \mathbf{M}, \\ & x_i \in \alpha_i, \\ & n\Delta \in \bar{\omega}(\pi_i(x), x) \\ & \quad \text{for } n \in \mathcal{Z}^+, \\ & \quad \pi_i(x) \neq \phi \\ \nu_i(x) & \text{otherwise} \end{cases}$$

Note that this construction permits states in which time has progressed past the point at which any edge transition could occur out of some given discrete state. This does not affect the reachable discrete state space. However, we do require that all integrators advance together whenever a Δ time unit passes. It is convenient to do this as part of an otherwise standard construction of a single automaton from the set of concurrent discrete time automatons.

The state space of the final finite-state discrete time automaton is $A_1 \times A_2 \times \cdots \times A_m \times \{0, \Delta, 2\Delta, \cdots, \mathbf{M}, \mathbf{M}+\Delta, \cdots \mathbf{M}+n\Delta\}^{|X|}$, where each system state consists of the discrete state of each process and the quantized value of each integrator. For each possible transition over a set of coedges in the discrete time concurrent system, there is a transition in the single automation that changes some of the discrete states. For each Δ transition, all integrator values increase using one of the transitions defined above for each integrator, i.e. every integrator x increases by some $n\Delta$ in the allowed rate interval. Another way to think of this is that the integrators behave like finite-state processes whose states are $\{0, \Delta, 2\Delta, \cdots, \mathbf{M}\}$, and all such processes synchronously perform Δ transitions. This is a finite-state automaton and so the discrete state reachability problem is decidable.

We will first make some useful observations about this automaton and then prove the theorem.

Figure 5 illustrates the relationship between the values an integrator takes on in the continuous time and the discrete time systems. The shaded zone indicates a set of points that can be reached from some starting point by all continuous time trajectories $x(t)$ whose derivatives stay within the specified bounds. The slopes of the upper and lower bounding lines are integral multiples of Δ. If the rate interval is singular then this region and its bounding lines are a single line.

In the discrete time system, transition times and integrator values at transitions are restricted to a set of mesh points $(i\Delta, j\Delta), i, j \in \mathcal{Z}^+$, illustrated by vertical and horizontal gray lines in Figure 5. Integrator slopes are restricted to integer multiples of Δ that fall within the specified rate intervals (always including the exact minimum and maximum rates), which are illustrated in the figure by three wide dark lines with slopes $\Delta, 2\Delta$ and 3Δ. The latticework of thin dark lines illustrate the piece-wise linear trajectories $x'(t)$ of the discrete time system, where $x'(t) \le \mathbf{M} + n\Delta$ for all x for some finite n in the discrete time system. Any of the mesh points that fall within the region bounded by the lines of maximum and minimum rate are reachable in the discrete time system. We refers to these regions as the rate-feasible regions for the continuous time and the discrete time systems, the latter being a set of mesh points.

The requirement that Δ be a divisor of $\{\frac{c_i}{\omega_j}, \cdots\}$ means that the line of slope ω_j that passes through the origin crosses the line $x = c_i$ at a mesh point. The intersection $c_i = \omega_j t$ occurs at time $t = \frac{c_i}{\omega_j}$, and since this is an integer multiple of Δ then so is t. Every c_j is also an integer multiple of Δ, and so the point (c_j, t) is a mesh point in the discrete time system. This is true for every rate interval end-point ω_j and every constraint interval end-point c_i. This is illustrated in Figure 5 by two dark gray lines labeled c_1 and c_2. We will call a horizontal line $x = c_i$ for some constraint end-point c_i a line of constraint.

This is also true for any line of slope ω_j that is displaced by an integer multiple of Δ from the origin, i.e. any line of slope ω_j that passes through a mesh point on the t axis also passes through a mesh point on every line of constraint $x = c_i$. The converse statement is also useful: through every mesh point on any line of constraint $x = c_i$ there is a line of slope ω_j for every ω_j that passes through a mesh point on the t axis and passes through a mesh point on every other line of constraint.

Suppose a line $x = \omega t$, with slope ω appearing in some rate interval in the model, passes through some mesh point on the t axis (and so through a mesh point when crossing every line of constraint). Given some point $x' = \omega t'$ on this line as illustrated in Figure 4, we can move backwards down the line to $(\lfloor t' \rfloor_\Delta, \lfloor x' \rfloor_\omega)$, the nearest mesh point below; and we can move forwards up the line to $(\lceil t' \rceil_\Delta, \lceil x' \rceil_\omega)$, the nearest mesh point above. Because this line only crosses lines of constraint at mesh points, all the points on the line segment between $(\lfloor t' \rfloor_\Delta, \lfloor x' \rfloor_\omega)$ and $(\lceil t' \rceil_\Delta, \lceil x' \rceil_\omega)$, and these two mesh points themselves, fall on, or on the same side of, all lines of constraint. For any nonsingular edge constraint, all of the points on this line seg-

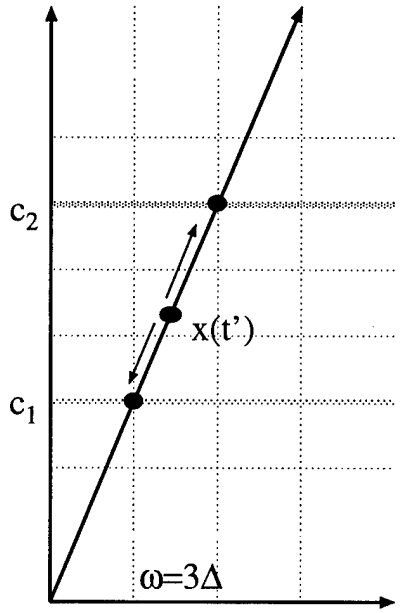


Figure 4: Mesh Points can be Reached Without Crossing Lines of Constraint

ment will satisfy that edge constraint if any of them do.

We now argue that a discrete state is reachable in a continuous time system that satisfies the assumptions of the theory iff that same discrete state is reachable in the corresponding discrete time system.

Half of this is easy to argue. Every trace of the discrete time system is also a trace of the continuous time system, so if a discrete state is reachable in the discrete time system then it is reachable in the continuous time system.

For the other half of the proof, we need to show that for every trace of a continuous time system there is a trace in the discrete time system that reaches the same discrete state. For this we argue that for every trajectory $x(t)$ in any continuous time trace there is a discrete time trajectory $x(t)'$ that is close enough to $x(t)$ to enable the same sequence of edge transitions in the same order in the discrete time system.

We are actually going to show something a bit stronger, that every continuous time trace is bounded both above and below by a discrete time trace. Consider the set of transitions occurring at times $t_{i_1}, t_{i_2}, \cdots$ in a time interval $[j\Delta, (j+1)\Delta]$ in the continuous time system, with integrator values $x_1(t_{i_n}), x_2(t_{i_n}), \cdots$ satisfying the edge constraints of each transition at time t_{i_n} (illustrated in Figure 6). We show that for each integrator the same set of transitions are enabled at both time $\lfloor t_i \rfloor_\Delta$ and at time $\lceil t_i \rceil_\Delta$ in the discrete time system. Since both options exist for every integrator, it is possible for all edge transitions in $[j\Delta, (j+1)\Delta]$ (each of which in general will constrain multiple integrators) to occur at either $j\Delta$ or $(j+1)\Delta$ in the same order as in the continuous time system.

We argue that for every integrator x and every transition at time t there is a mesh point at $\lfloor t \rfloor_\Delta$ and a mesh point at $\lceil t \rceil_\Delta$ that falls within the rate-feasible region of the discrete time automaton and satisfies the same edge constraints as the value $x(t)$. We argue this by induction over the sequence of transitions along each trajectory $x(t)$, assuming it is true for $x(t_i)$ when proving it is true for $x(t_{i+1})$. We argue this in three cases: first, when $\bar{\omega}(x)$ is singular and no constraint $(x, \bar{c})$ is singular; second, when neither $\bar{\omega}(x)$ nor any constraint $(x, \bar{c})$ are singular; and finally, when $\bar{\omega}(x)$ is not singular but some constraints $(x, \bar{c})$ may be singular.

Figure 7 illustrates the argument for the first case, when $\bar{\omega}(x)$ is singular but there are no singular constraints $(x, \bar{c})$. In the basis case, both systems start with $x(0) = 0$ and identical values for $\bar{\omega}(x)$, so the rate-feasible regions are identical. For any transition at time t_i the values $x(\lfloor t_i \rfloor_\Delta)$ and $x(\lceil t_i \rceil_\Delta)$ are mesh points of the discrete time system. As noted earlier, every line of slope $\bar{\omega}(x)$ crosses every applicable line of constraint at a mesh point. This condition, illustrated by the horizontal lines labeled c_1 and c_2 in the figure, will be maintained as an induction invariant. As noted earlier we may reach, but cannot cross over, any applicable constraint end-point by following $x(t)$ down to the mesh point $x(\lfloor t_i \rfloor_\Delta)$ or up to the mesh point $x(\lceil t_i \rceil_\Delta)$. Consequently, both of these mesh points also satisfy the same set of edge constraints as $x(t_i)$ in the continuous time system.

The preceding argument establishes the basis case, and in fact holds for all transitions up to the first point at which $\pi(x)$ is set to ϕ or at which x is reset to 0. Whenever this occurs, however, the value of the continuous time $x(t)$ always falls between two reachable and feasible piece-wise lines in the discrete time system, as illustrated in Figure 7. For if $\pi(x)$ becomes null at some time t_i then the discrete time zero-slope line from $x(\lfloor t_i \rfloor_\Delta)$ must lie below and the discrete time zero-slope line from $x(\lceil t_i \rceil_\Delta)$ must lie above the zero-slope line from $x(t_i)$ in the continuous time system. If $\pi(x)$ becomes non-null at some later time $t_j \geq t_i$ a similar argument holds. The continuous time line from $x(t_j)$ now falls between some pair of discrete time lines separated by a distance of Δ, and so there is always a discrete time mesh point on one of these lines

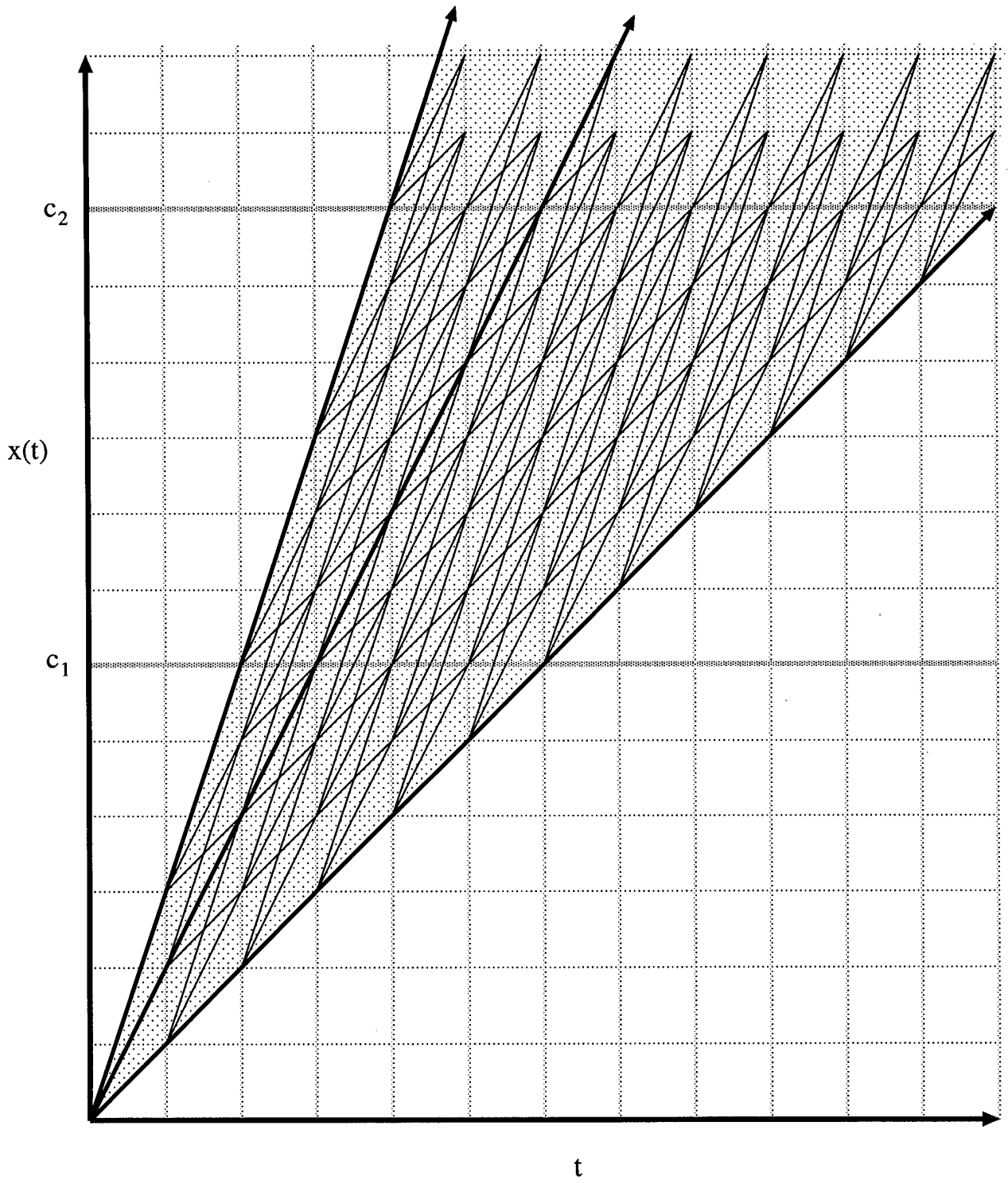


Figure 5: Spaces Reachable by $x(t)$ in Continuous vs Discrete Systems

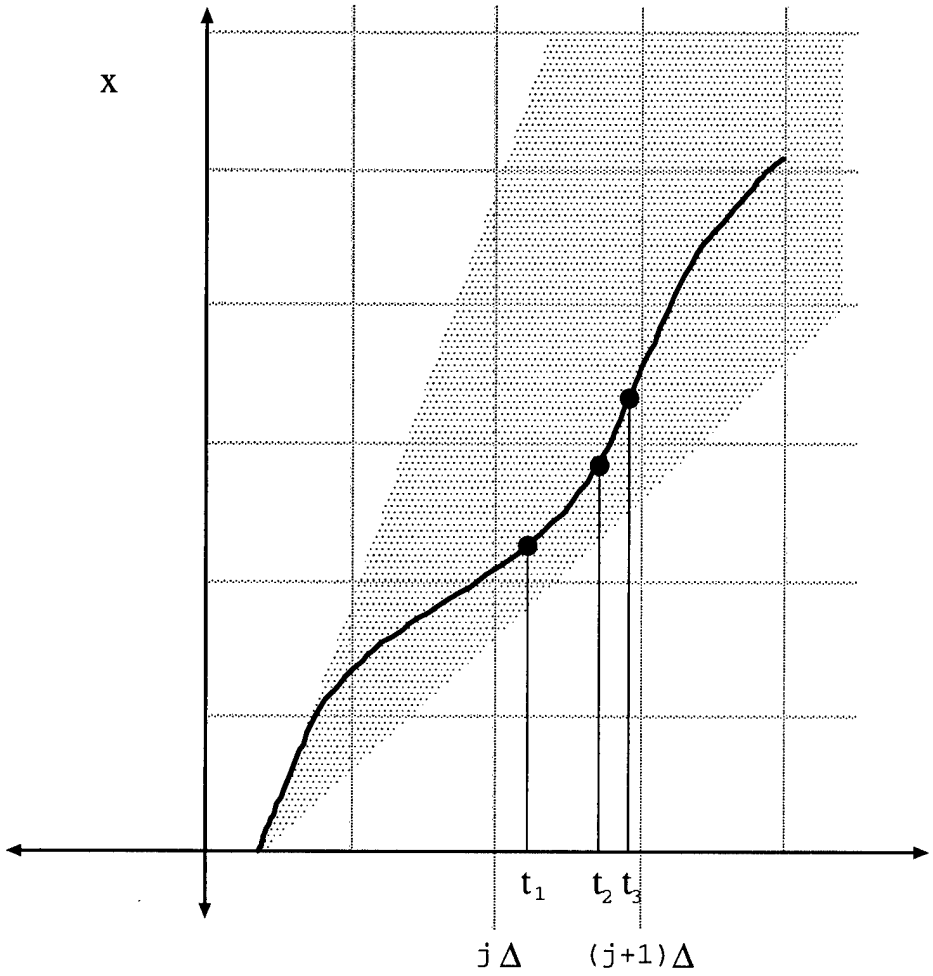


Figure 6: A Continuous $x(t)$ with Three Transitions in $[j\Delta, (j+1)\Delta]$

in any interval $[\lfloor t_k \rfloor_\Delta, \lceil t_k \rceil_\Delta]$ in which some successor transition at time $t_k \geq t_j$ can occur. All these possible discrete time trajectories are being nondeterministically explored by the discrete time automaton. A similar argument holds when $\pi(x) = \phi$ initially (becoming non-zero later) and when x is reset.

When $\pi(x)$ becomes non-null or when x is reset, the upwardly-sloping lines from the mesh points on either side of $x(t)$ in the discrete time system are always parallel to the previous line segments of slope $\omega(x)$ and are displaced to the right by an integer multiple of Δ. As explained earlier, this means these lines still have the property that they cross lines of constraint only at mesh points, and so the assumptions of the induction are preserved for subsequent transition points.

Figure 8 illustrates the argument for the second case, when neither $\bar{\omega}(x)$ nor any constraint $(x, \bar{c})$ is singular. We apply the argument of the preceding case to the lines defined by the lower and upper rates of $\bar{\omega}(x)$, the lines that define the boundary of the continuous time rate-feasible region. Since both the upper and lower bounding lines of the continuous time rate-feasible region are contained by rate-feasible lines in the discrete time system, the entire continuous time rate-feasible region is contained within the discrete time rate-feasible region for each integrator. For any continuous $x(t)$ through the continuous time rate-feasible region, for any point $x(t_i)$ at time t_i, there is always some near-by mesh point in the discrete time rate-feasible region at $\lfloor t_i \rfloor_\Delta$, and one at $\lceil t_i \rceil_\Delta$, where both mesh points fall on the same side of every horizontal line defined by an applicable constraint endpoint.

Figure 9 illustrates the argument for the third case, when a constraint interval $(x, \bar{c})$ may be singular providing $\bar{\omega}(x)$ is not. We again argue by induction over

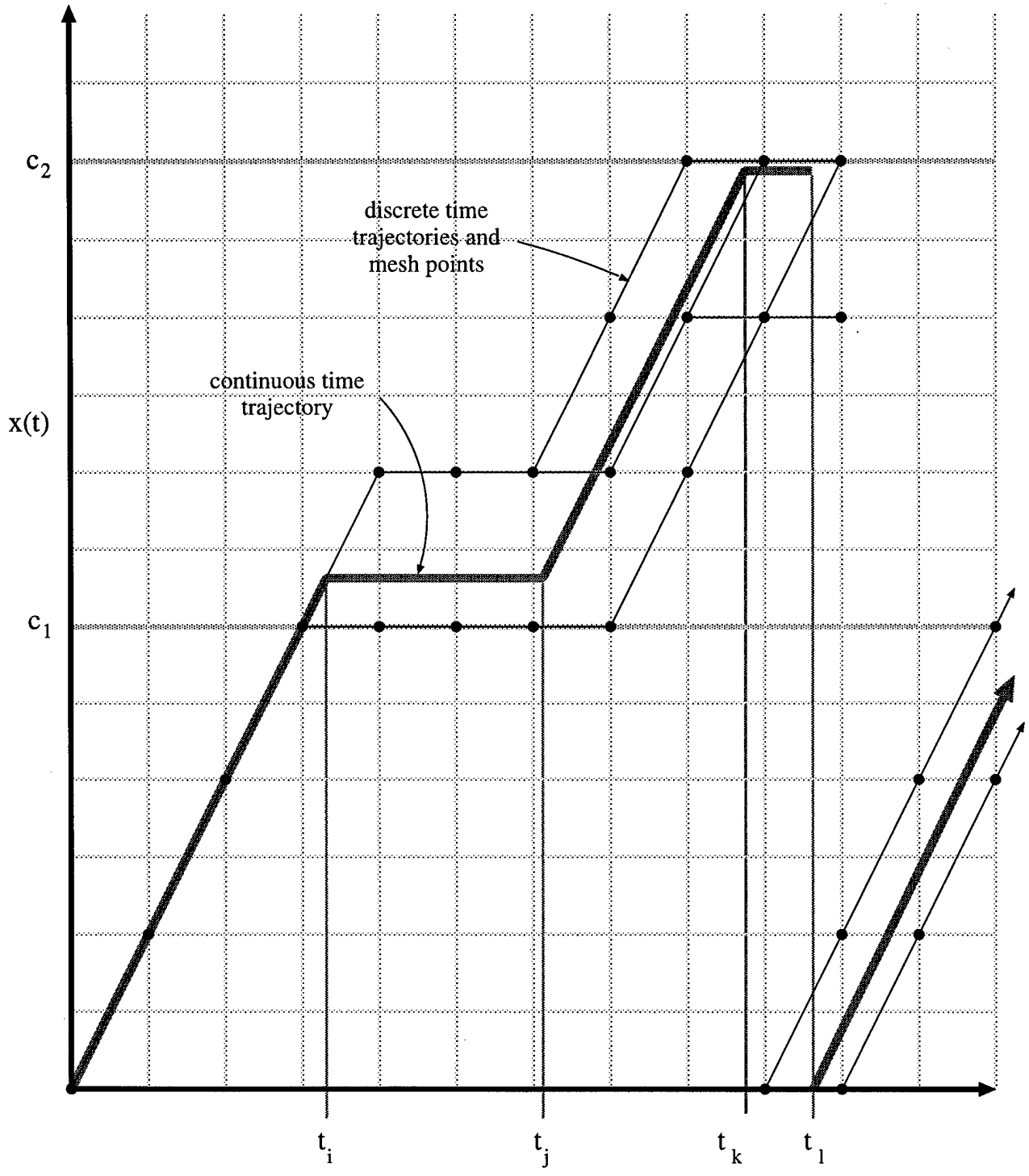


Figure 7: Case when $\bar{\omega}(x)$ but no $(x, \bar{c})$ is singular

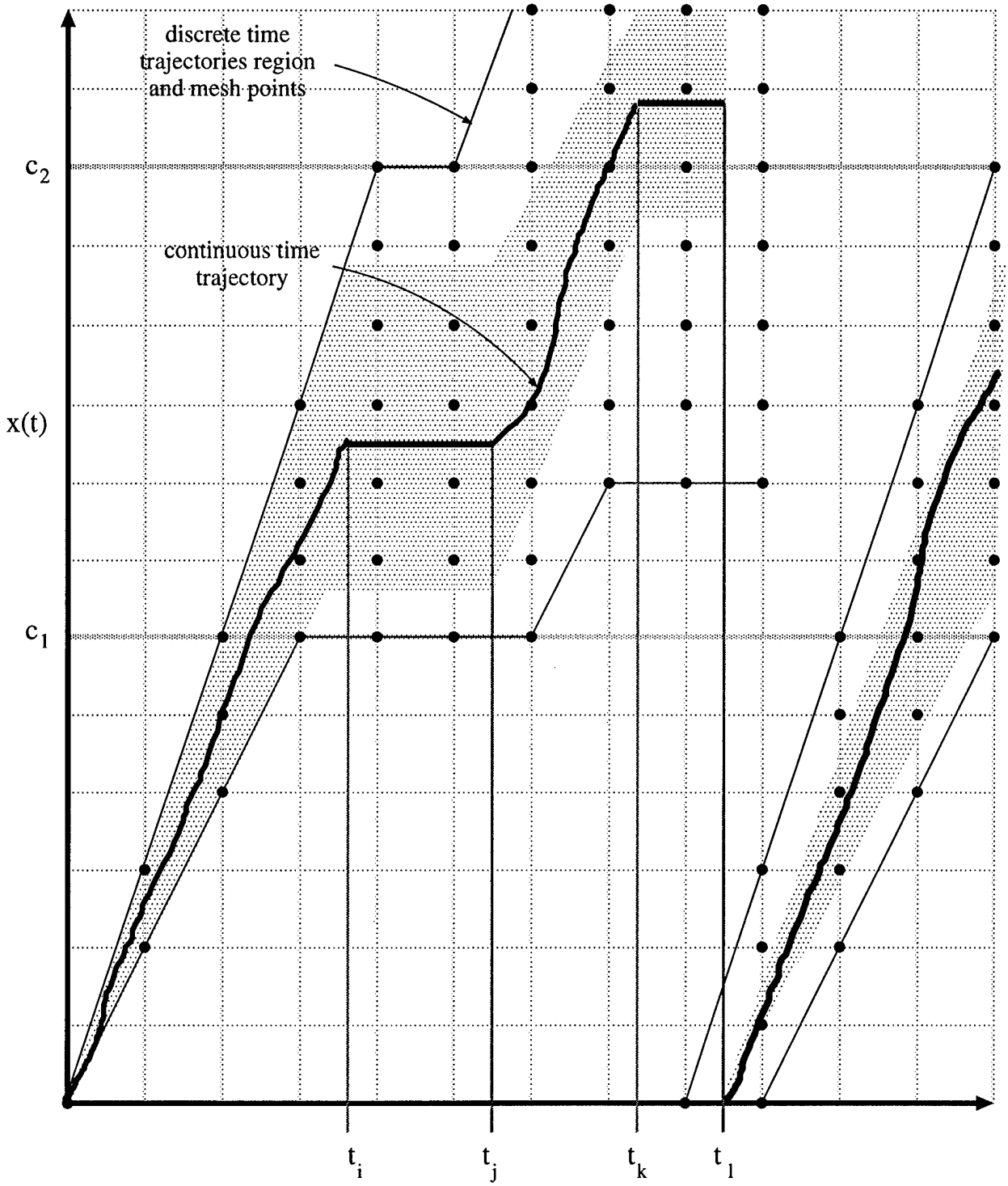


Figure 8: Case when $\bar{\omega}(x)$ and $(x, \bar{c})$ are not singular

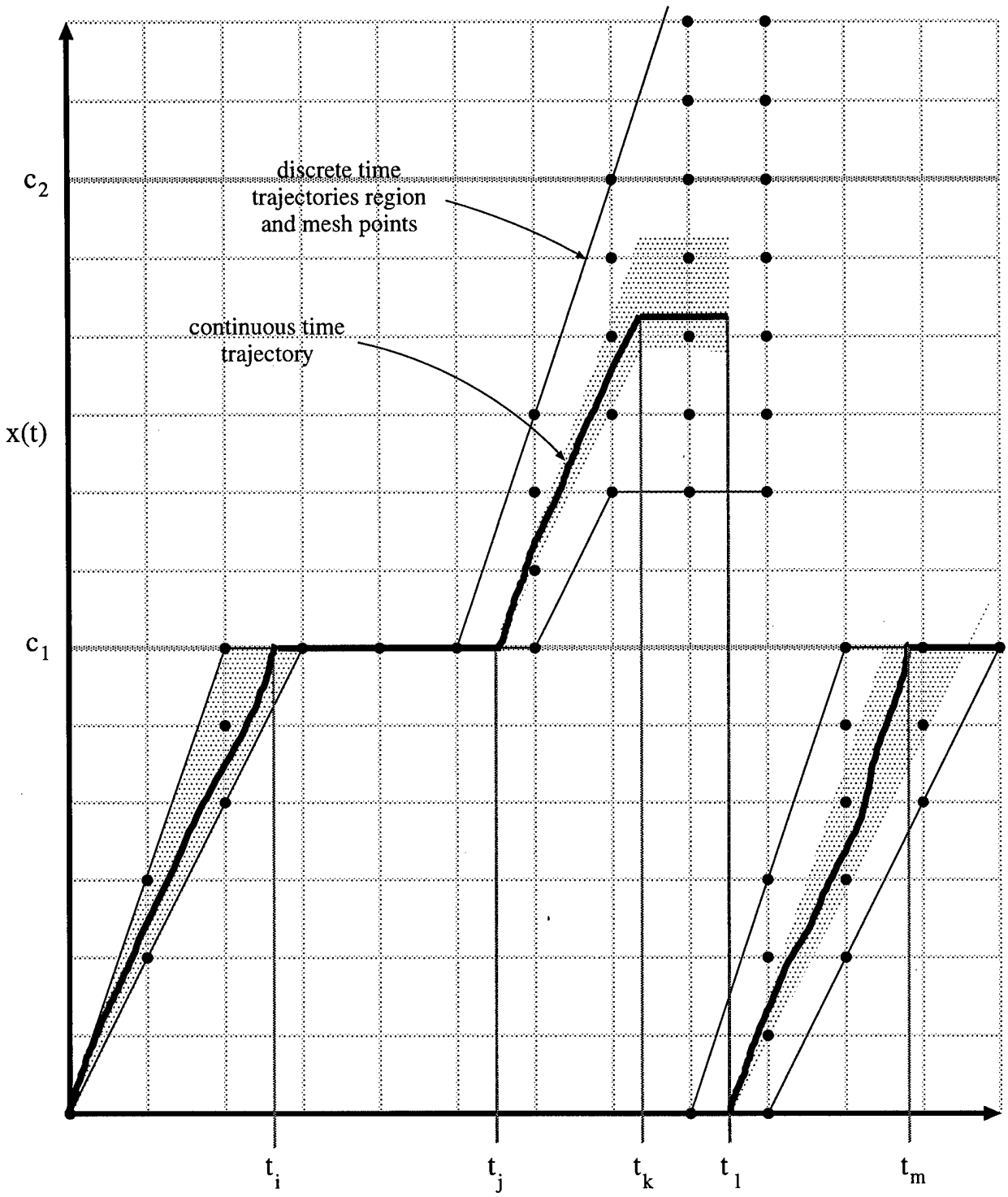


Figure 9: Case when no $\bar{\omega}(x)$ but some $(x, \bar{c})$ are singular

the transition points along the continuous time trajectory $x(t)$. For the basis, recall that lines of maximum and minimum rate passing through mesh points on the t axis are equal to applicable lines of constraint only at mesh points. If $\bar{\omega}(x)$ is non-singular then these points of intersection must be separated by at least Δ at the lowest line of constraint. We maintain as an induction invariant that the two lines bounding the rate-feasible region in the discrete time case pass through either the same or adjacent mesh points on the t axis, which means they always pass through nonadjacent mesh points on every applicable line of constraint. This insures that whenever $x(t)$ is exactly equal to any line of constraint, the two mesh points on either side (which satisfy the same equality constraint) are reachable in the discrete time system. The existence of two discrete time mesh points on either side of any $x(t)$ when it crosses a line of constraint thus holds for all transitions starting from the initial state up to the point at which $x(t)$ is reset or changes slopes from something in $\bar{\omega}(x)$ to 0 or back again.

Either a continuous time $x(t)$ is reset or changes slope to 0 or back between two lines of constraint or exactly on a line of constraint. If the former, the argument of the second case applies and there are always two rate-feasible discrete time lines satisfying the induction assumptions on either side of any continuous time $x(t)$. If the latter, then $x(t)$ in the continuous time system falls along the line of constraint and passes through rate-feasible mesh points until the slope of $x(t)$ becomes non-zero again. As illustrated in Figure 9, the discrete time system explores trajectories from both mesh points on either side of a continuous time transition that occurs when $x(t)$ equals a line of constraint and that resets $x(t)$ or changes its slope.

The conditions of the final exception force the continuous time trajectories for the constrained integrators to also be discrete time trajectories. Requiring the rate interval $\bar{\omega}(x)$ to be singular forces $x(t)$ to consist of line segments between transition events. The restrictions on edge constraints only allow $x(t)$ to be reset or to change slope at mesh points. For any transitions occurring at continuous time t' along edges that impose constraints on such integrators, either one of the applicable constraints is singular in which case $t\prime$ is a discrete time; or none of the applicable constraints are singular in which case the transition could occur at either of $\lfloor t\prime \rfloor_\Delta$ or $\lceil t\prime \rceil_\Delta$ as argued previously for case one above.

6 Remarks

Our decidability result leaves two obvious open questions.

First, this result only allows dynamic reallocation between processors of different rates when a process is dispatched and its accumulated compute time variable reset to zero.

Second, scheduling disciplines that rely on comparisons of accumulated compute times are not supported. Edge guards of the form $x \leq y + c$ would allow decisions to be based on comparisons involving accumulated compute times. We note such guards can be used in timed automata, which are decidable[2, 9].

References

[1] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.

[2] Rajeev Alur and David Dill, "The Theory of Timed Automata," in J.W. de Bakker, C. Huizing, W.P. de Roever and G. Rozenberg (Eds.), *Real Time: Theory in Practice*, LNCS 600 (Springer-Verlag, 1992).

[3] Rajeev Alur and David L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science* 126, 1994.

[4] Rajeev Alur and Thomas A. Henzinger, "Logics and Models of Real Time: A Survey," *Real Time: Theory in Practice*, J.W. de Bakeer, K. Huizing, W.-P. de Roever and G. Rozenberg, eds., *Lecture Notes in Computer Science*, Springer-Verlag, 1992.

[5] Rajeev Alur, Tomás Feder and Thomas A. Henzinger, "The Benefits of Relaxing Punctuality," *Proceedings of the Tenth Annual ACm Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 19-21, 1991.

[6] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell and Andy J. Wellings, "Fixed Priority Pre-emptive Scheduling: An Historical Perspective," *Real-Time Systems*, 8, 1995.

[7] Patrice Brémond-Grégoire and Insup Lee, "A Process Algebra of Communicating Shared Resources with Dense Time and Priorities," University of Pennsylvania Department of Computer Science Technical Report MS-CIS-95-08, June 1996.

[8] S. Campos, E. Clarke, W. Marrero and M. Minea, "Computing Quantitative Characteristics

of Finite-State Real-Time Systems," Department of Computer Science, Carnegie Mellon University.

[9] David L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," *International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 12-14, 1989, also in *Lecture Notes in Computer Science 407*, J. Sifakis (Ed.), Springer-Verlag, pp 197-212.

[10] Andre N. Fredette and Rance Cleaveland, "RSTL: A Language for Real-Time Schedulability Analysis," *Proceedings of the Real-Time Systems Symposium*, December 1993.

[11] Andre N. Fredette, *A Generalized Approach to the Analysis of Real-Time Computer Systems*, Ph.D. Dissertation, North Carolina State University, March 1993.

[12] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal of Applied Mathematics*, Vol. 17, No. 2, March 1969.

[13] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri and Pravin Varaiya, "What's Decideable About Hybrid Automata?" *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995, `http://www-cad.eecs.berkeley.edu/~tah/Publications/whats_decidable_about_hybrid_automata.html`

[14] M. G. Harbour, M. H. Klein and J. P. Lehoczky, "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority," *IEEE Real-Time Systems Symposium*, December 1991.

[15] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[16] Y. Kesten, A. Pnueli, J. Sifakis and S. Yovine, "Integration Graphs: A Class of Decideable Hybrid Systems," in R. L. Grossman, A. Nerode, A. P. Ravn and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, Springer-Verlag, 1993.

[17] Insup Lee, Patrice Brémond-Grégoire and Richard Gerber, "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," Department of Computer Science, University of Pennsylvania.

[18] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, 20(1), January 1973.

[19] Gavin Lowe, "Specification and Proof of Prioritized, Timed CSP," Oxford University Computing Laboratory Technical Report PRG-TR-17-92, `http://www.comlab.ox.ac.uk/oucl/publications/tr/TR-`

[20] Nancy A. Lynch and Mark R. Tuttle, "An Introduction to Input/Output Automata," MIT Technical Memo MIT/LCS/TM-373, also *CWI Quarterly*, 2(3), September 1989.

[21] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[22] Jin Yang, Aloysius K. Mok and Farn Wang, "Symbolic Model Checking for Event-Driven Real-Time Systems," *ACM Transactions on Programming Languages and Systems*, v19, n2, March 1997.

A New Linear Hybrid Automata Reachability Procedure

– Draft of March 14, 2000 –

Steve Vestal
steve.vestal@honeywell.com
Honeywell Technology Center
Minneapolis, MN 55418*

Abstract

We present a new on-the-fly method for reachability analysis of linear hybrid automata with constant rates. The novelty of our methods lie in the algorithms used to manipulate polyhedra, and in our encoding of domain-specific behavior (real-time scheduling in our case) into the model semantics. Our polyhedra operations can be performed in polynomial time, typically quadratic in the number of continuous variables in a model. Encoding scheduling into the model semantics allows us to significantly reduce the size of the overall system discrete state space. We benchmarked a prototype of our method, HyTech, and Verus using a randomly generated set of classical real-time uniprocessor workloads. We also experimented with two optimization methods, a simplification of model parameters using results from real-time scheduling theory, and a simple form of partial order reduction. When we started our work using HyTech we were able to consistently analyze workloads having 4 concurrent tasks (81 reachable discrete states), using our prototype together with these optimization methods we were able to consistently analyze workloads having 13 concurrent tasks (8192 reachable discrete states).

1 Introduction

Linear hybrid automata are finite state automata augmented with variables whose values change continuously in a way that depends on the current discrete state. The values of the continuous variables can affect, and can be affected by, discrete transitions between discrete states. Linear hybrid automata can be subjected to a reachability analysis to verify that a given set of assertions is true of a system. In general, a semi-decision procedure must be used since the reachability problem for linear hybrid automata is undecidable[14] (as opposed to timed automata, which are decideable[3]). It can be shown, however, that reasonable pragmatic restrictions make models for real-time allocation and scheduling in distributed heterogeneous systems decideable[23].

Linear hybrid automata can be used to model many kinds of dynamical systems, but the problem of particular interest to us is the modeling and schedulability analysis of real-time systems. The continuous variables of a linear hybrid automaton can be used as timers to control task dispatching and detect missed deadlines, and as so-called integration variables to record accumulated task compute time. Linear hybrid automata are sufficiently powerful to model a number of interesting system features, such as remote procedure calls, rendezvous between tasks, variations in compute time as a function of internal task state, and distributed synchronization and communication protocols. Reasonably detailed models of source code can be written, and linear hybrid automata are also useful for verifying implementations of such things as time-dependent protocols and scheduling kernels[24].

The reachable state space for a linear hybrid automaton is a set of regions, where each region consists of a discrete state plus a polyhedron that defines a set of possible values for the continuous variables. We present a reachability procedure that represents polyhedra as systems of linear inequalities. We present new algorithms for computing the polyhedron that results when time is allowed to pass and variable values change at specified rates; and the polyhedron that results when a variable is unconstrained and removed from the system. These algorithms might be viewed as generalizations of the difference methods used for timed automata[8, 3] and exhibit a vague similarity to the pragmatic algorithm used earlier for quantifier elimination[2]. We present an algorithm to decide if one polyhedron is contained in another. We present a reduction algorithm to simplify the set of constraints that represent a polyhedron. Our prototype is an on-the-fly tool that enumerates regions as they are encountered, rather than first enumerating the complete reachable discrete state space and then enumerating the reachable polyhedra for each discrete

*This work has been supported by the Air Force Office of Scientific Research under contract F49620-97-C-0008.

state. However, our procedure restricts variable rates to be specified constants in each discrete state and does not provide parametric analysis[15].

We also discuss different ways that real-time scheduling can be incorporated into a linear hybrid automata model, including a novel way of adding scheduling semantics to create an extended resourceful linear hybrid automata model. Extending the semantics of the model rather than trying to write a standard linear hybrid automata model of a scheduling protocol both reduces the size of the region space and allows a much broader range of scheduling protocols to be modeled.

We randomly generated a sequence of uniprocessor workloads consisting of periodic and aperiodic tasks scheduled using preemptive fixed priority. For each of these we generated a linear hybrid automata model whose assertions were satisfied only for schedulable task sets. These models were analyzed using a prototype implementation of our procedure. All these task sets were amenable to analysis using the exact characterization algorithm[18], which we used to double-check our results. We submitted the same models to HyTech[15], a linear hybrid automata tool; and to Verus[7], a discrete timed automata tool.

We also experimented with two optimization methods. First, traditional uniprocessor preemptive priority scheduling theory says that we can replace execution time and event inter-arrival intervals with their worst-case values. Second, we experimented with a simple partial order reduction method.

The earliest reachability tool of which we are aware, HyTech, represented polyhedra as finite sets of linear constraints[2]. The operations performed on these polyhedra used quantifier elimination, a formal way to algebraically manipulate and make decisions about systems of linear inequalities in which some of the variables are existentially quantified. Polka and a later version of HyTech used a pair of representations, the traditional system of linear inequalities together with polyhedra generators consisting of sets of vertices and rays[12, 15]. Different operations required during reachability are more convenient in the different representations, and methods are used to convert between the two as needed. These previous methods are subject to the theoretical risk that some polyhedra operations may require a combinatorial amount of time, although we did not test for this in our experiments. Our polyhedra operations are all doable in polynomial time (although we used the Simplex algorithm in our prototype), typically quadratic in the number of constraints used to represent a polyhedron.

A variety of differences between the tools and certain aspects of our use of them make direct comparisons questionable, and we experimented only with a particular class of problem. Keeping these caveats in mind, we were able to solve problems with our prototype tool an order of magnitude more quickly than with HyTech, which was perhaps three orders of magnitude faster than Verus without automatic variable reordering. Perhaps as importantly, our prototype tool was more numerically robust and used significantly less memory, it never failed due to numeric overflow or memory exhaustion. We were able to solve problems that HyTech and Verus could not solve. When we began our work using HyTech we were able to consistently solve systems of 4 tasks having 81 reachable discrete system states. Using our prototype tool together with some experimental optimization methods, we were able to consistently solve systems of 13 tasks having 8192 reachable discrete states. In our judgement this is not yet adequate for schedulability analysis but is at the threshold of utility for simple but practical verification problems[24]. Our work suggests that future improvements could result in further significant increases in the size of solvable problem, and we discuss this in our concluding section.

2 Resourceful Hybrid Automata

A hybrid automaton is a finite state machine augmented with a set of real-valued variables and a set of propositions about the values of those variables. Figure 1 shows an example of a hybrid automaton whose discrete states are `preempted`, `executing` and `waiting`; and whose real-valued variables are c and t. `Waiting` is marked as the initial discrete state, and c and t are assumed to be initially zero.

Each of the discrete states has an associated set of differential equations, e.g. $\dot{c} = 0$ and $\dot{t} = 1$ for the discrete state `preempted`. While the automaton is in a discrete state, the continuous variables change at the rates specified for that state.

Edges may be labeled with guards involving continuous variables, and a discrete transition can only occur when the values of the continuous variables satisfy the guard. When a discrete transition does occur, designated continuous variables can be set to designated values as specified by assignments labeling that edge.

A discrete state may also be annotated with an invariant constraint to assure progress. Some discrete transition must be taken from a state before that state's invariant becomes false. For example, the hybrid automaton in Figure 1 must transition out of state `computing` before the value of c exceeds 100.

The hybrid automata of interest to us are called linear hybrid automata because the invariants, guards and assignments are all expressed as sets of linear constraints. The differential equations governing the continuous dynamics in a particular linear hybrid automaton discrete state are restricted to the form $\dot{x} \in [l, u]$

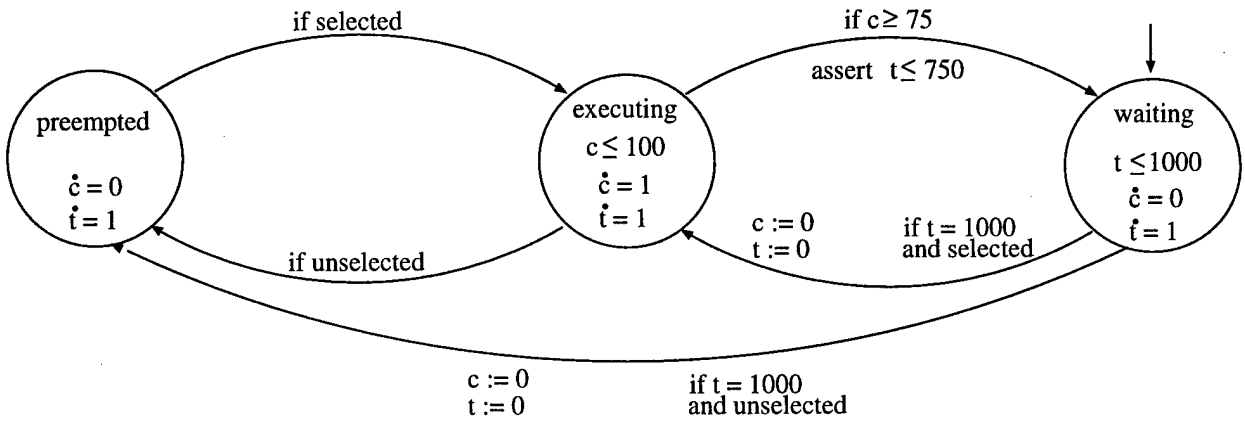


Figure 1: A Hybrid Automata Model of a Preemptively Scheduled Task

where $[l, u]$ is a fixed constant interval. Our method further restricts this to a singleton rate, $\dot{x} = i$.

We want to verify assertions about the behavior of a hybrid automaton. Although it is possible in general to check temporal logic assertions[2], we make do by annotating discrete states and edges with sets of linear constraints labeled as assertions. These constraints must be true whenever the system is in a discrete state or whenever a transition occurs over an edge.

The cross-product construction used to compose concurrent finite state processes can be extended in a fairly straight-forward way to systems of hybrid automata. The invariant and assertion associated with a discrete system state are the conjunction of the invariants and assertions of the individual discrete states. The guards, assertions and assignments of synchronized transitions are the conjunction and union of the guards, assertions and assignments of the individual discrete co-edges. If there is a conflict between the rate assignments of individual discrete states, or a conflict between the variable assignments of co-edges, then the system is considered ill-formed. Note that concurrent hybrid automata may interact through shared real-valued variables as well as by synchronizing their transitions over co-edges.

The application of interest in this paper is the analysis and verification of real-time systems. Figure 1 shows an example of a simple hybrid automata model for a preemptively scheduled, periodically dispatched task. A task is initially waiting for dispatch but may at various times also be executing or preempted. The variable t is used as a timer to control dispatching and to measure deadlines. The variable t is set to 0 at each dispatch (each transition out of the waiting state), and a subsequent dispatch will occur when t reaches 1000. The assertion $t \leq 750$ each time a task transitions from executing to waiting (each time a task completes) models a task deadline of 750 time units. The variable c records accumulated compute time, it is reset at each dispatch and increases only when the task is in the computing state. The invariant $c \leq 100$ in the computing state means the task must complete before it receives more than 100 time units of processor service, the guard $c \geq 75$ on the completion transition means the task may complete after it has received 75 time units of processor service (i.e. the task compute time is uncertain and/or variable but always falls in the interval $[75, 100]$).

In this example the edge guards `selected` and `unselected` represent scheduling decisions made at scheduling events (often called scheduling points in the real-time literature). These decisions depend on the available resources (processors, busses, etc.) being shared by the tasks. There are several approaches to introduce scheduling semantics into a model having several concurrent tasks.

Scheduling can be introduced using concepts taken from the theory of discrete event control[20]. A concurrent scheduler automaton can be added to the system of tasks. The scheduling points in the task set become synchronization events at which the scheduler automaton can observe the system state and make control decisions. Many high-level concepts from discrete event control theory carry over into this domain, such as the importance of decentralized control and limited observability in distributed systems.

Discrete event control theory provides an approach to synthesize optimal controllers, which in this domain translates to the automatic construction of application-specific scheduling algorithms. However, classical discrete event control theory does not deal with time. The theory has been extended to synthesize nonpreemptive schedulers for timed automata[4, 1], but this excludes preemptively scheduled systems. It is possible to develop scheduling automata by hand using traditional real-time scheduling policies such as preemptive fixed

priority. Examples have been given in the literature, where each distinct ready queue state is modeled as a distinct discrete state of the scheduler automaton[2]. This would allow a very large class of scheduling algorithms to be modeled, but the size of the scheduler automaton may grow combinatorially with the number of tasks.

It is possible to model preemptive fixed priority scheduling by encoding the ready queue in a variable rather than in a set of discrete states. A queue variable is introduced that will take on only integer values. At each transition where a task i is dispatched, 2^i is added to this queue variable; at each transition where task i completes, 2^i is subtracted. The queue variable can be interpreted as a bit vector whose i^{th} bit is set whenever task i is ready to compute. There is no separate scheduler automaton, the scheduling protocol is modeled using additional guards and states in the task automata. This is the approach we took when we started our work using HyTech. This encodes a specific scheduling protocol into each task model, and adds additional discrete states, variables and guards to the model. It is awkward to model any scheduling policy other than simple preemptive fixed priority without inheritance.

In the end, we found it simpler and more general to define a slightly extended linear hybrid automata model that includes resource scheduling semantics[23]. The discrete state composition of the task set is performed before any scheduling decisions are made. A scheduling function is then applied to the composed system discrete state to determine the variable rates to be used for that system state. In essence, the composed system discrete state is the ready queue to which the scheduling function is applied, very much analogous to the way run-time scheduling algorithms are applied in an actual real-time system. It is not necessary to have different discrete states for preempted and computing, since this information is now captured in the variable rates. It is not necessary to model a scheduling algorithm as a finite state control automaton added to the system, it is not necessary to encode a specific scheduling semantics into the task automata. One simply codes up a scheduling algorithm in the usual way and links it with the rest of the reachability analysis code. This approach significantly reduces the number of discrete states in the model and simplifies the modeling of the desired scheduling discipline. The formal details of this model and its semantics are recorded elsewhere[23].

3 Reachable Regions

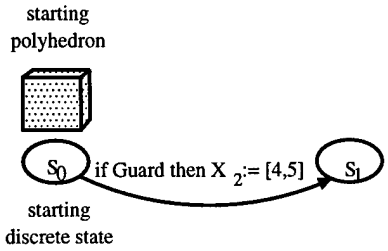
A state of a linear hybrid automaton consists of a discrete part, the discrete state at some time t; and a continuous part, the real values of the variables at time t. It turns out that, although this state space is uncountably infinite, the reachable state space for a given linear hybrid automaton is a subset of the cross-product of the discrete states with a recursively enumerable set of convex polyhedra in $\Re^n$ (where n is the number of variables)[2]. A region of a linear hybrid automaton is a pair consisting of a discrete state and a convex polyhedron, where convex polyhedra can be represented using a finite set of linear constraints. Model checking consists of enumerating the reachable regions for a given linear hybrid automaton and checking to see if they satisfy the assertions.

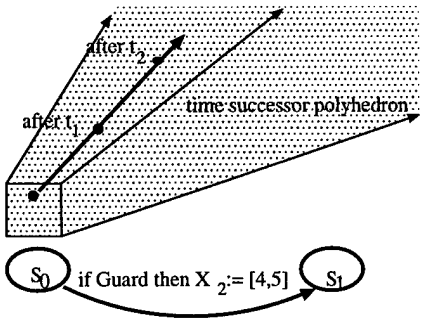
Figure 2 depicts the basic sequence of operations that, given a starting region (a discrete state and a polyhedron defining a set of possible values for the variables), computes the set of values the variables might take on in that discrete state as time passes; and computes a set of regions reachable by subsequent discrete transitions.

The first step is the computation of the time successor polyhedron from the starting polyhedron (often called the post operation). For each point in the starting polyhedron, the time successor of that point is a line segment beginning at that point whose slope is defined by the variable rates specified for the discrete state. This is the set of variable values that can be reached from a starting point by allowing some amount of time to pass. The time successor of the starting polyhedron is the union of the time successor lines for all points in the starting polyhedron. A basic result of linear hybrid automata theory is that the time successor of any convex polyhedron is itself a convex polyhedron (which in general will be unbounded in certain directions)[2].

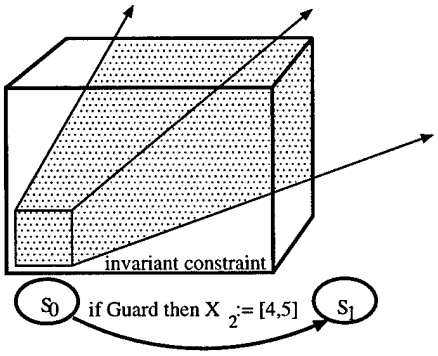
The second step is the intersection of the time successor polyhedron with the invariant constraint associated with the discrete state. Polyhedra are easily intersected by taking the union of the set of linear constraints that define the two polyhedra. This is the time successor region that is feasible given the invariant specified for the discrete state.

The remaining steps are used to compute new regions reachable from this feasible time successor region by some transition over an edge. For each edge out of the current discrete state, the associated guard is first intersected with the feasible time successor region. This polyhedron, if nonempty, defines the set of all variable values that might exist whenever the discrete transition could occur. Any variable assignments associated with the edge must now be applied to this polyhedron. This is done in two phases. First, a variable to be assigned a new value $x := l$ is unconstrained (often called the free operation). This operation leaves unchanged the relationships between all other variables, i.e. the polyhedron is projected onto the subspace $\Re^{n-1}$ of the remaining variables. This result is then intersected with

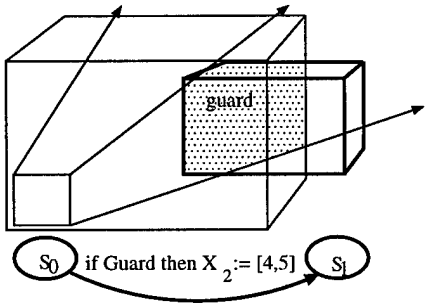




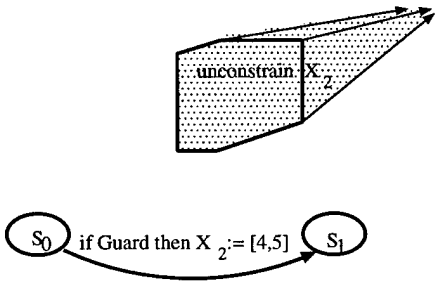
Step 1: Time Successor



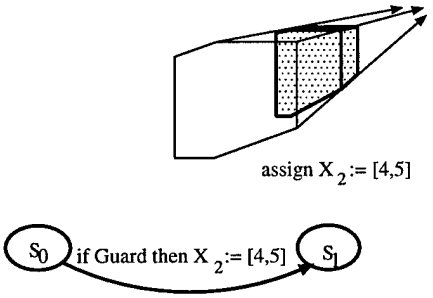
Step 2: Intersection with Invariant



Step 3: Intersection with Guard



Step 4: Unconstrain Assigned Variable



Step 5: Intersect with Assignment Constraint

Figure 2: Hybrid Automata Reach Forward Operations

the constraint $x = l$. This polyhedron, together with the discrete state to which the edge goes, is a new region for which the above steps may be repeated. In general a set of assignments whose right-hand sides are linear formula are allowed, with some restrictions. The variables to be assigned are unconstrained and the resulting polyhedra are then intersected with the appropriate linear constraints in some order. With care, fairly complex sequences of assignments can be modeled on a single edge[24].

The overall method begins at the initial region of a hybrid automaton. The operations described above are applied to enumerate feasible time successor regions and the new regions reachable from these via discrete tran-

sitions. As new regions are enumerated, they must be checked to see if they have been visited before (otherwise the method will not terminate even when there are a finite number of regions). This is done by comparing the discrete states of regions for equality, and by checking to see if the new polyhedron is contained in the polyhedron of a previously visited region. This is summarized in Figure 3.

4 Operations on Polyhedra

In the following descriptions we use X to denote a set of real-valued variables $x_1, x_2, ...x_n$, $\dot{X}$ to denote an assignment of integer rates to these variables, $e = c_1x_1 + c_2x_2 + ... + c_nx_n$ a linear formula over X with integer coefficients, and $\dot{e}$ the rate or derivative of this formula given some $\dot{X}(\dot{e} = c_1\dot{x_1} + c_2\dot{x_2} + ... + c_n\dot{x_n})$. We use $\bar{r} \in \Re^n$ to denote some specific point in $\Re^n$, a specific assignment of values to X. We use $\bar{r} \uparrow \delta$ as a shorthand notation for $\bar{r} + \dot{X}\delta$, the point reached from $\bar{r}$ after allowing δ time to pass given variable rates $\dot{X}$. We use P to denote a set of constraints and the phrase "polyhedron P" to refer to the set of all solutions to the system of constraints P. We sometimes abuse notation and write $\bar{r} \in P$ to mean the point $\bar{r}$ is in the polyhedron P, $\bar{r}$ satisfies the system of constraints P.

4.1 Time Successor

The time successor of a set of constraints given constant rate $\dot{X}$ is computed in two steps.

1. Let each constraint $l_i \leq e_i \leq u_i$ where $\dot{e_i} \neq 0$ be written so that $\dot{e_i} > 0$, which can be achieved by multiplying the constraint by -1 if needed. For each distinct pair of constraints

$$l_i \leq e_i \leq u_i$$
$$l_j \leq e_j \leq u_j$$

where $\dot{e_i} > 0$ and $\dot{e_j} > 0$, add to the set the constraint

$$\dot{e_j}l_i - \dot{e_i}u_j \leq \dot{e_j}e_i - \dot{e_i}e_j \leq \dot{e_j}u_i - \dot{e_i}l_j$$

2. Replace each constraint $l \leq e \leq u$ where $\dot{e} > 0$ by $l \leq e \leq \infty$.

The rate of each constrained expression added in the first step is 0, so the second step only applies to constraints that were in the original polyhedron. The number of operations required by the algorithm is quadratic in the number of constraints whose expressions have non-zero rate.

Theorem: Let P be a satisfiable set of constraints, and let P' be computed from P using the above algorithm. Then

- For each $\bar{r} \in P$, for each $\delta \geq 0$, $\bar{r} \uparrow \delta \in P'$ (allowing any amount of time to pass for any point that satisfies P yields a point that satisfies P').
- For each $\bar{r}' \in P'$ there exists some $\delta \geq 0$ such that $\bar{r} \downarrow \delta \in P$ (every point in P' is reachable from some point in P by allowing some amount of time to pass).

Proof: We show the first part by demonstrating that none of the constraints modified or introduced by the algorithm are invalidated by allowing time to pass from any initial value $\bar{r} \in P$. Step 1 adds constraints that are already implied by existing constraints in P, so these are all satisfied by any $\bar{r} \in P$. Applying step 1 results in a polyhedron that has the same solution set as P. After step 2, every constrained expression with zero rate appears unchanged in P', every constrained expression with positive rate is unbounded so that no amount of time can make the modified constraint infeasible. This proves the first part of the theorem.

We show the second part by demonstrating that $\bar{r}' \in P'$ implies there exists some $\delta \geq 0$ such that $\bar{r}' \downarrow \delta \in P$. Observe that every constraint $l \leq e \leq u$ where $\dot{e} = 0$, including every constraint added in step 1, appears in both P and P'. The values of the constrained expressions, and hence the feasibility of these constraints, remain unchanged as time passes. What we need to show is that every $\bar{r}'$ that satisfies the set of constraints in P' that were loosened to $l_i \leq e_i \leq \infty$ in step 2 can be taken back in time by some $\delta \geq 0$ to a point $\bar{r} \downarrow \delta$ that satisfies the original constraints. That is, we need to show there exists a single value $\delta \geq 0$ such that all of

$$l_i \leq e_i - \dot{e_i}\delta \leq u_i$$

are feasible for every point $\bar{r}' \in P'$. We can rewrite these constraints as

$$\begin{aligned} & l_i \leq e_i - \dot{e_i}\delta \leq u_i \\ \equiv \quad & l_i - e_i \leq -\dot{e_i}\delta \leq u_i - e_i \\ \equiv \quad & e_i - l_i \geq \dot{e_i}\delta \geq e_i - u_i \\ \equiv \quad & \tfrac{e_i - l_i}{\dot{e_i}} \geq \delta \geq \tfrac{e_i - u_i}{\dot{e_i}} \end{aligned} \tag{1}$$

There exists a value of $\delta \geq 0$ that satisfies all these inequalities when evaluated at $\bar{r}'$ if there exists a value for δ that simultaneously falls between the upper and lower bounds of all these constraints. This can occur when no lower bound exceeds any upper bound, which can occur when the set of constraints

$$\tfrac{e_i - l_i}{\dot{e_i}} \geq \tfrac{e_j - u_j}{\dot{e_j}}$$

is feasible for all pairs i and j. For $i = j$ this reduces to $u_i \geq l_i$, always true when P is feasible. For $i \neq j$ we can rewrite these as

$$\dot{e_j}e_i - \dot{e_i}e_j \geq \dot{e_j}l_i - \dot{e_i}u_j$$

```
Entry_Region,
New_Region: Region;
Successor_Polyhedron,
Constrained_Polyhedron,
Guarded_Polyhedron,
Unconstrained_Polyhedron,
Assigned_Polyhedron: Polyhedron;
Examined,
To_Be_Examined: set of Region;

Examined := empty;
To_Be_Examined := initial region;
while Not_Empty (To_Be_Examined) loop
   Entry_Region := Choose_And_Remove_One_Of (To_Be_Examined);
   Add_To_Set (Examined, Entry_Region);
   Successor_Polyhedron := Time_Successor (Entry_Region.Polyhedron);
   Constrained_Polyhedron := Intersect (Successor_Polyhedron, Entry_Region.Discrete_State.Invariant);
   Check_Region_Assertion (Entry_Region.Discrete_State, Constrained_Polyhedron);
   for each Transition in Entry_Region.Discrete_State.Transitions_From loop
       Guarded_Polyhedron := Intersect (Constrained_Polyhedron, Transition.Guard);
       if Not_Empty (Guarded_Polyhedron) then
           Check_Transition_Assertion (Transition, Guarded_Polyhedron);
           Unconstrained_Polyhedron := Unconstrain (Guarded_Polyhedron, Transition.Assignments.Variables);
           Assigned_Polyhedron := Intersect (Unconstrained_Polyhedron, Transition.Assignments);
           New_Region.Discrete_State := Transition.To_Discrete_State;
           New_Region.Polyhedron := Assigned_Polyhedron;
           for each Previous_Region in Union (Examined, To_Be_Examined) loop
              if Contained_In (New_Region, Previous_Region) then
                 goto next Transition loop;
              end if;
           end loop;
           Add_To_Set (To_Be_Examined, New_Region);
       end if;
   end loop;
end loop;
```

Figure 3: High-Level Region Enumeration Procedure

These were added to P' in step 1 and, since the rate of their constrained expressions is 0, remain unchanged by step 2 in P'. These constraints are thus satisfied for every $\bar{r}' \in P'$, so there exists a $\delta \geq 0$ that satisfies the constraints in (1), and this δ is such that $\bar{r}' \downarrow \delta \in P$.

4.2 Unconstrain

To unconstrain a variable x we must remove all constraints that contain that variable. However, there may be constraints between other variables that are transitively implied by a set of removed constraints. For example, $l_1 \leq y - x \leq u_1$ and $l_2 \leq x - z \leq u_2$ imply $l_1 + l_2 \leq y - z \leq u_1 + u_2$, this information must be preserved before removing the constraints involving x. We unconstrain a variable x in a set of constraints P by constructing a new set of constraints P' using the following steps.

1. Let each constraint $l \leq e \leq u$ in P where e has an instance of x be written in the form $l \leq cx - e' \leq u$, where e' involves the remaining variables and their coefficients and $c > 0$. For each distinct pair of such constraints in P

$$l_i \leq c_i x - e_i \leq u_i$$
$$l_j \leq c_j x - e_j \leq u_j$$

combine the two in a way that cancels the x terms, adding to P' the constraint

$$c_j l_i - c_i u_j \leq c_i e_j - c_j e_i \leq c_j u_i - c_i l_j$$

2. Each constraint $l \leq e \leq u$ where e has no instances of variable x is added to P'.

Let $X \setminus x$ refer to the set of variables X minus the variable x, and let $\bar{r} \setminus x$ for $\bar{r} \in \Re^n$ refer to the $n-1$ vector of values that are identical to $\bar{r}$ with the value for variable x removed.

Theorem: Let P be a feasible set of difference constraints, and let P' be computed from P by applying the above algorithm to unconstrain the variable x. If $\bar{r} \in P$ then $\bar{r} \setminus x \in P'$, and if $\bar{r} \setminus x \in P'$ then there exists some value for variable x such that $\bar{r} \in P$.

Proof: We will show that $\bar{r} \in P$ iff $\bar{r} \setminus x \in P'$ for some value for x.

Each constraint added to P' in step 1 is implied by some pair of constraints in P. Constraints added in step 2 are the same in P and P'. Consequently, every constraint in P' is implied by one or two constraints in P, and every value $\bar{r} \in P$ thus satisfies the constraints in P'. To state this another way, the constraints added to P' are never any tighter than constraints that occur in P. Note that this means P' is feasible since P is assumed feasible.

For the second part of the proof, consider a value $\bar{r} \setminus x \in P'$. We need to show there exists some value for variable x such that $\bar{r}$ satisfies all of the constraints

$$l_i \leq c_i x - e_i \leq u_i \tag{2}$$

in P. There will exist a value for x that satisfies the above constraints if there is a value for x that simultaneously satisfies every pair of constraints

$$\frac{l_i+e_i}{c_i} \leq x \leq \frac{u_i+e_i}{c_i}$$
$$\frac{l_j+e_j}{c_j} \leq x \leq \frac{u_j+e_j}{c_j}$$

where $c_i, c_j > 0$. For $i = j$ this reduces to $l_i \leq u_i$, which always holds when P is feasible. For $i \neq j$ we can rewrite these as

$$\begin{aligned} & c_j l_i + c_j e_i \leq c_i u_j + c_i e_j \\ \equiv \quad & c_j l_i - c_i u_j \leq c_i e_j - c_j e_i \end{aligned}$$

These were all added to P' in step 1 and are all satisfied for every $\bar{r} \setminus x \in P'$ (P' is feasible as noted above). There thus exists a value for x that satisfies the constraints in (2), and this value is such that $\bar{r} \in P$.

4.3 Intersection

The intersection of the solution sets for two systems of constraints P_1 and P_2 is the set of values that satisfies both systems of constraints, which is the set of solutions to $P_1 \bigcup P_2$ (the union of the sets of constraints has as its solution the intersection of the corresponding polyhedra). This can be done in linear time.

4.4 Feasibility

The feasibility of a set of inequalities P can be determined as a side-effect of solving the associated linear programming problem with some trivial objective function, e.g. $\max \sum_i x_i$ given P for the variables x_i appearing in P.

We note that feasibility testing seems fundamentally as hard as linear programming[6]. Each linear programming problem has an associated dual problem of the same size, with the property that only optimal solutions are feasible for both. Thus, any feasibility test capable of identifying a feasible solution can be used to solve a linear programming problem by applying that test to the union of the constraints of the original problem and its dual.

We discovered by experiment that guessing a set of variable values (the mid-point of each rectangular constraint) then evaluating the constraints using those values was an effective approximate test. This test can quickly confirm that certain polyhedra are feasible, and our experiments suggest that perhaps half of all feasibility tests could be resolved using this method. However, feasibility testing accounted for a relatively small portion of the overall execution time of our prototype, and this approximation had no siginificant impact.

4.5 Containment

Given two sets of constraints I (inner) and O (outer) we want to determine if every solution to I is also a solution to O (whether the polyhedron I is contained in the polyhedron O). We do this using the following algorithm.

1. If it can be (quickly) determined that I and O do not intersect, then I cannot be contained in O.
2. If O contains variables that do not appear in I then terminate with a negative result.
3. For each constraint $l \leq e \leq u$ in O solve the linear programming problems $\bar{x}_l = \min e$ given I and $\bar{x}_u = \max e$ given I. If $l \leq \bar{x}_l$ and $u \geq \bar{x}_u$ for every constraint in O then polyhedron I is contained in polyhedron O. Otherwise, the algorithm terminates with a negative result when the first constraint from O is found that does not pass this test.

The first step is a prefilter to efficiently detect certain common cases where I obviously cannot be contained in O. We did not do an exact feasibility test, only an approximate one that quickly checks to see if the intersection is definitely infeasible (discussed later). Our experience suggests that over 80% of all containment tests are resolved by this prefilter. This was important for our prototype, which spent most of its time searching for containing polyhedra (the loop for each `Previous_Region` in Figure 3).

To establish the correctness of the final step we prove the following.

Theorem: For each constraint $l \leq e \leq u$ in O let $\bar{x}_l = \min e$ given I and $\bar{x}_u = \max e$ given I be solutions to linear programming problems. Every feasible value for I is also a feasible value for O iff $l \leq e(\bar{x}_l)$ and $u \geq e(\bar{x}_u)$ for every constraint $l \leq e \leq u$ in O.

Proof: Suppose $l \leq e(\bar{x}_l)$ and $u \geq e(\bar{x}_u)$ for every constraint $l \leq e \leq u$ in O. It is known that optimal values for the objective function of a linear programming problem are achieved at some boundary vertex or facet of the polyhedron I (or else the value of the objective function

is unbounded). For $\bar{x}_l = \min e$ given I the value of e at every point in the polyhedron is bounded below by $e(\bar{x}_l)$, and for $\bar{x}_u = \max e$ given I the value of e at every point in the polyhedron is bounded above by $e(\bar{x}_u)$ (or else the value of e is unbounded below or above, respectively). If $l \leq e(\bar{x}_l)$ and $u \geq e(\bar{x}_u)$ then every point in the polyhedron I satisfies the constraint $l \leq e \leq u$ in O (where we allow l or u to be $-\infty$ or ∞ respectively). If this holds for all constraints in O then every feasible value for I also satisfies all the constraints of O.

4.6 Assertion Checking

An assertion A where A is a system of linear constraints can be evaluated for a given polyhedron P by seeing if P is contained in A. Conjunctions and disjunctions of sets of linear inequalities can be evaluated in the obvious way.

4.7 Reduction

The time successor and unconstrain operations may cause a quadratic increase in the number of constraints. An essential element of our procedure is the use of an algorithm to reduce the number of constraints used to represent a polyhedron by identifying and eliminating redundant constraints. We combine a fast but approximate bounds tightening procedure with a more effective but expensive Simplex-based procedure to detect and eliminate redundant constraints.

An important part of our procedure is the initial use of an efficient bounds tightening procedure to simplify sets of constraints[5]. For each pair of constraints

$$\begin{aligned} l_i \leq e_i \leq u_i \\ l_j \leq e_j \leq u_j \end{aligned}$$

if there exist integers $c_i, c_j > 0$ such that $c_i e_i = c_j e_j = e$ then these constraints can be replaced by the single constraint

$$\max(c_i l_i, c_j l_j) \leq e \leq \min(c_i u_i, c_j u_j)$$

There are two such rules, one for $c_i > 0$ and one for $c_i < 0$ (c_j can always be made positive). Our prototype implementation maintains polyhedra as lists of constraints that are lexicographically sorted by variable, so that any two linearly dependent constraints will appear adjacent in the list. This bounds tightening operation is applied during each intersection operation (implemented as a linear-time merge of two sorted lists), including each time a new constraint is added to a constraint list.

The time successor and unconstrain operations add constraints that are differences of existing constraints, and in practice many of these added constraints are redundant with each other. These two algorithms frequently add triplets of the form

$$\begin{aligned} l_{ij} \leq e_i - e_j \leq u_{ij} \\ l_{jk} \leq e_j - e_k \leq u_{jk} \\ l_{ik} \leq e_i - e_k \leq u_{ik} \end{aligned} \tag{3}$$

where the third constraint may be implied by the sum of the first two (ignoring constant multipliers). Similarly, the first constraint of the triplet

$$\begin{aligned} l_{ij} \leq e_i - e_j \leq u_{ij} \\ l_j \leq e_j \leq u_j \\ l_i \leq e_i \leq u_i \end{aligned} \tag{4}$$

may be implied by the difference of the other two constraints. We can check for these implications and use them to tighten constraints and eliminate redundant constraints. This could be viewed as an approximate generalization of the shortest path algorithm used to simplify bounded difference matrices[8, 3].

More precisely, for each triplet of constraints

$$\begin{aligned} l_i \leq e_i \leq u_i \\ l_j \leq e_j \leq u_j \\ l_k \leq e_k \leq u_k \end{aligned}$$

if there exist integers $c_i, c_j, c_k > 0$ such that $c_i e_i - c_j e_j = c_k e_k$ then constraint $l_k \leq e_k \leq u_k$ can be replaced by

$$\min(c_k l_k, c_i l_i - c_j u_j) \leq c_k e_k \leq \max(c_k u_k, c_i u_i - c_j l_j)$$

There are four such rules, one for each combination of possible signs for c_i and c_j (c_k can always be made positive). Linear dependence is transitive in the sense that when dependence is detected, each of the three can be tightened using similar formulas involving the other two.

Checking all possible triplets would be $O(n^3)$ in the number of constraints. Instead, we record with most constraints two references T_i and T_j to two other constraints with whose sum it is likely to be linearly dependent. For each constraint added by the time successor operation, T_i and T_j are the two constraints differenced to form that constraint. For each constraint added by the unconstrain operation, we search for constraints that have the same variables as the two that were differenced except for the variable being unconstrained. Our experience suggests that reasonable T_i and T_j can be identified for most constraints involving two or more variables.

Let (e, T_i, T_j) be a constraint e and its associated T_i, T_j references. We iterate over all triplets of constraints $(e_1, T_i, T_j), (e_2, T_j, T_k), (e_3, T_i, T_k)$ to produce candidates likely to have the form shown previously in (3). We also iterate over all triplets $(e, T_i, T_j), T_i, T_j$. Each triplet is checked for linear dependence, which can be determined by solving a simple 2×2 system of linear

equations involving the coefficients of variables common to the three constraints. Where linear dependence is detected, each of the constraints has its bounds tightened using the bounds implied by the appropriate linear combination of the other two constraints.

In our prototype we associate with each constraint in the list its index or numeric position in the list. The pair (T_i, T_j) is kept in a canonical order where the index of T_i is less than the index of T_j. We produce a list of references to the constraints that is lexicographically sorted by the index values for (T_i, T_j). Using this sorted list, it is possible to iterate over triplets of constraints in quadratic time.

In our prototype the constraint list is ordered so that all constraints involving the same variables are adjacent to each other. Each time we check a triplet of constraints e_i, e_j, e_k for linear dependence, we check all triplets having the same variables as the set e_i, e_j, e_k and not just those three individual constraints. Our experience suggests this heuristic is worth the additional cost, which tends to be relatively small since subsequences of constraints involving identical sets of variables tend to be relatively short.

In general, multiple such iterations may be needed to find a fixed point at which no constraint bounds are tightened any further. Our experience suggests this can in fact be limited to a small fixed number, such as 3, without any significant impact.

As constraints are tightened, a note is made for each bound as to whether that bound is implied by other nonredundant constraints on the list. For example, if $l_i \leq e_i \leq u_i$ and $e_i = e_j + e_k$ and $l_i = l_j + l_k$ then the lower bound l_i is implied by the other two constraints and is redundant, providing $l_j \leq e_j$ and $l_k \leq e_k$ are not marked as redundant. A final pass is performed to replace all redundant lower and upper bounds by $-\infty$ and ∞ respectively. All constraints where both the lower and upper bounds are redundant are deleted (our implementation always retains the tightest rectangular constraint for every variable appearing in a polyhedron for book-keeping reasons).

If, during any bounds tightening operation, $u < l$ for any constraint $l \leq e \leq u$ then the system is definitely not feasible. This feasibility test is not exact, it is possible for $l \leq u$ for every constraint in an infeasible system of constraints. However, we use this as a fast test to quickly detect many cases of infeasibility, including the prefilter for our containment test. Our experience suggests that about $\frac{1}{3}$ of all feasibility tests can be decided in this manner, or about 80% if the containment prefilter is also counted.

If the above bounds tightening procedure fails to reduce the number of constraints in a polyhedron to less than half the average number of constraints in all polyhedra, then our prototype tool applies a more expensive but more effective procedure. For each constraint $l \leq e \leq u$ in P solve the linear programming problems $\bar{x}_l = \min e$ given $P - \{l \leq e \leq u\}$ and $\bar{x}_u = \max e$ given $P - \{l \leq e \leq u\}$. If $l \leq \bar{x}_l$ and $u \geq \bar{x}_u$ then $l \leq e \leq u$ is redundant and is removed. Our experience suggests that over 95% of all reductions are performed using the bounds tightening procedure alone, but it is nevertheless essential to include this more effective procedure. Without this, our prototype was sometimes unable to complete an analysis due to the presence of a few polyhedra for which bounds tightening was ineffective, where these few polyhedra formed a brick wall that prevented complete reachability analysis.

5 Implementation Notes

We implemented our prototype in a compiled language, Ada 95. We represented constraint bounds and coefficients using 64 bit integers. We used a sparse vector representation that only stores non-zero coefficients. Polyhedra were represented as doubly linked lists of constraints. The set of reached regions was stored by hashing the discrete system state, then storing a list of polyhedra for each discrete system state.

Our primal/dual Simplex algorithm used double precision floating point and a sparse matrix representation. The Simplex algorithm is used only as a decision procedure, it does not compute any values that appear in any polyhedra. Nevertheless, this is a notable theoretical shortcoming in our prototype, which would ideally use rational arithmetic for the Simplex procedure. In this application large numbers of polyhedra have degenerate vertices, and our experience suggests that the Simplex implementation must include methods to deal with degeneracy[9, 17].

Efficient and robust computation of greatest common divisors (GCDs) during polyhedra reduction proved interesting and important enough to merit some comment. If two numbers are represented as products of their prime factors $X = 2^{x_1}3^{x_2}5^{x_3}...$ and $Y = 2^{y_1}3^{y_2}5^{y_3}...$ then the exponents in the prime factorization of their GCD is the min of the exponents of the two values[11],

$$\mathrm{GCD}(X, Y) = 2^{\min(x_1,y_1)}3^{\min(x_2,y_2)}5^{\min(x_3,y_3)}...$$

When computing the prime factorization of Y for the purpose of obtaining a GCD, it is only necessary to determine the exponents out to the last non-zero exponent of X, e.g. when computing $\mathrm{GCD}(12, Y)$ it is only necessary to compute the prime factors y_1 and y_2. This is because all the remaining prime factor exponents for 12 are 0, and $\min(0, y_k) = 0$ for any y_k. When computing the GCD of all numbers appearing in a constraint we first sort those numbers in ascending order, which

greatly reduces the need to determine the exponents of the larger prime factors of the larger numbers. However, we still encounter constraints having very large and relatively prime values, too large to be factored using any reasonably sized table of prime numbers. In this case we apply Euclid's algorithm to adjacent pairs of numbers in the sorted list in a way that reduces the length by half, e.g. $\mathrm{GCD}(X_1, X_2), \mathrm{GCD}(X_3, X_4), \ldots$. This increases the likelihood the two numbers submitted to Euclid's algorithm have about the same magnitude. This halved list of sorted numbers is then processed recursively.

A new polyhedron is added to the list of regions for a discrete state only when it is not contained in an existing region. However, the new polyhedron might contain a previously visited polyhedron. We check for this condition for all polyhedra that are still on the to-be-examined list and remove any polyhedra that are contained in the newly added one. We access the to-be-examined list in first-in-first-out order (depth-first search), which seems to result in a slightly higher percentage of to-be-examined polyhedra being removed than when using a first-in-first-out order (breadth-first search).

6 Benchmark Results

To exercise our prototype we randomly generated a series of 100 linear hybrid automata models for traditional uniprocessor workloads consisting of repetitively dispatched noninteracting harmonic tasks. Each task had minimum and maximum period and compute time values whose magnitudes reflected a reasonable level of real-world precision (e.g. period of 400 with a compute time range of [61, 73]). Aperiodic tasks were distinguished from periodic tasks by having unequal minimum and maximum periods. Although schedulability in this case is known to be a function of the maximum compute time only, we generated both minimum and maximum compute times because this is significant in distributed systems and verification problems, and this affects the set of reachable regions. Tasks were scheduled using a deadline monotonic preemptive fixed priority discipline.

We wrote a translator to the input specification language for HyTech, a linear hybrid automata reachability tool[2, 15]. Each task had four discrete states: computing, waiting, preempted, and rescheduling. In addition to the timer and accumulated compute time variables for each task, we introduced a single integer-valued queue variable whose n^{th} bit is set when the n^{th} task is enqueued. Using this queue variable and a reschedule synchronization event, we were able to specify preemptive fixed priority scheduling in a compositional model of the system. Models were analyzed using the `-o2` option to reduce the incidence of numeric overflow.

With reasonable practical restrictions, continuous time hybrid automata models of fairly complex real-time scheduling and allocation problems can be reduced to equivalent and decideable discrete time models[23]. We also wrote a translator to the input specification language for Verus, a discrete timed automata reachability tool[7]. Scheduling was performed by introducing an additional task for this purpose, as described in the literature and in examples that come with the tool. We extended the default number of bits to 20, with a corresponding increase in the number of boolean state variables. We were unable to use the `-r` option to automatically reorder variables, which is cited as being almost essential to achieve good performance. The reason is that this option is normally used as models are developed and grow incrementally, our initial experimentation suggests that the time required to do this from scratch for a full-blown model greatly exceeds the time required to analyze the model without automatic variable reordering.

We generated two variants of this set of models. One set included both feasible and infeasible problems. We used this set to check that all these tools, plus a traditional exact characterization schedulability analysis algorith[22], agreed. We generated another set that included only feasible problems. This set forced all tools to explore the entire reachable region space and was used for benchmarking purposes.

The ability of these tools to solve the generated feasible models is summarized in Figure 4. This figure shows the percentage of models that were solved as a function of the number of tasks. We imposed a time limit of 1 hour and a memory limit of 300 megabytes all tools. Tool failures also occurred due to numeric overflow and other problems.

Figure 5 shows the solution times in seconds as a function of the number of tasks for those models that were solved by all of the tools at each plotted point, using a logarithmic scale. That is, the figure does not include solution times for models that were solved by our prototype but not by HyTech. We include both individual problem solution times and a line showing the average solution times. The solution times for a fixed number of tasks (a fixed number of variables and reachable discrete states) vary significantly because the number of regions can vary significantly due to even small changes in the values of numeric parameters such as task periods.

We do not believe our results are sufficient to conclude there is any inherent superiority of continuous over discrete time models. BDD techniques are sensitive to variable ordering, and there may be a predictable variable ordering for problems of this particular type that yields significantly better performance. There are generalizations of BDD, such as IDD, that may have better

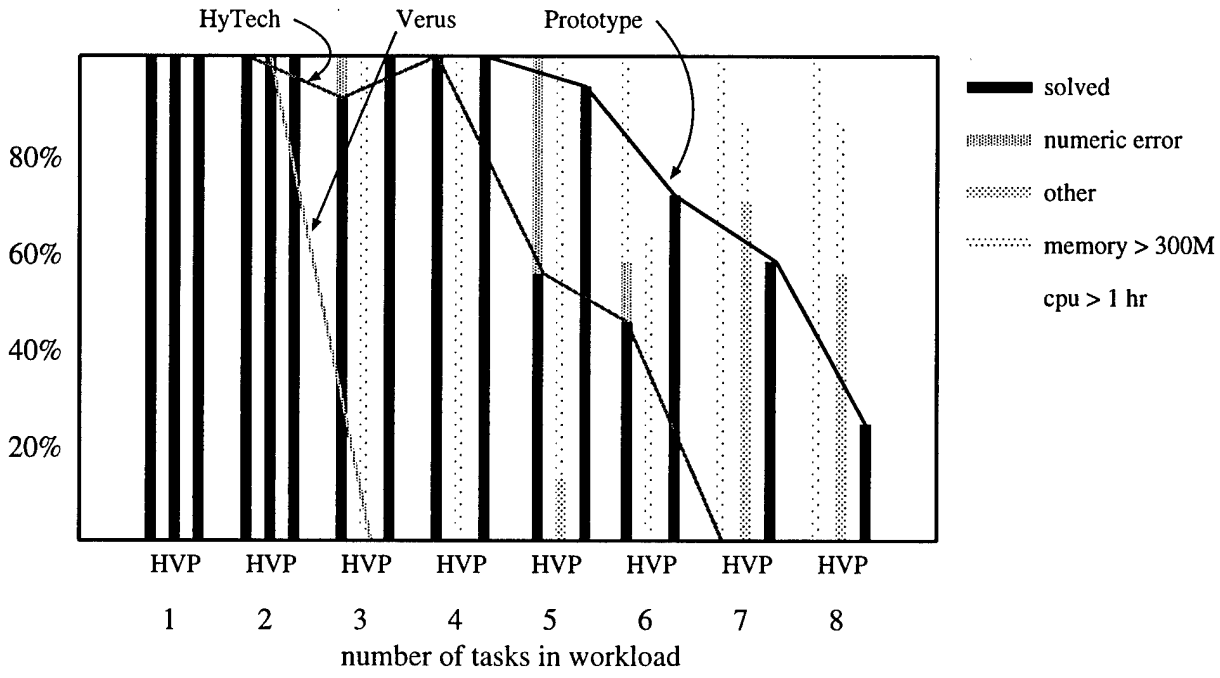


Figure 4: Percentage of Generated Problems That Were Solved

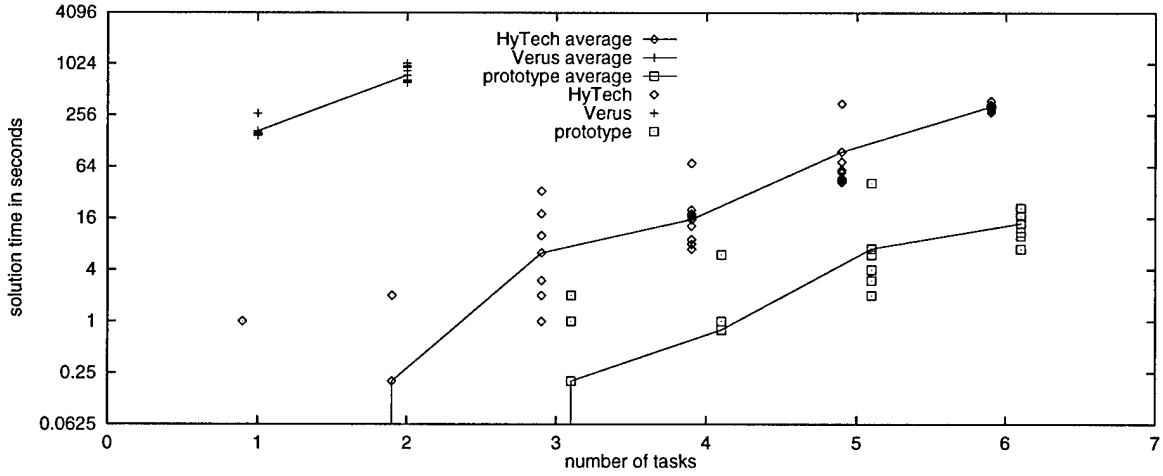


Figure 5: Solution Times for Problems That Were Solved

performance[21].

It is difficult to do a direct comparison between the methods we employ and those found in HyTech because there is no one single difference.

Our polyhedra operations are restricted to constant fixed rates. HyTech allows rate ranges to be specified and supports parametric analysis.

On this particular set of benchmarks, our prototype used less memory and was more numerically robust. However, these symptoms may not be due to fundamental differences, they might both be local and easily fixed artifacts of the current HyTech implementation. Improved numeric robustness might also be due to our use of a floating point Simplex implementation, a theoretically questionable aspect of our current prototype.

HyTech first enumerates the reachable discrete state space then enumerates the polyhedra. We do on-the-fly reachability, which may suppress enumeration of some discrete system states for some problems because edge guards may prevent transitions that would otherwise occur in the purely discrete model. However, in our set of benchmarks all of the discrete states were reached

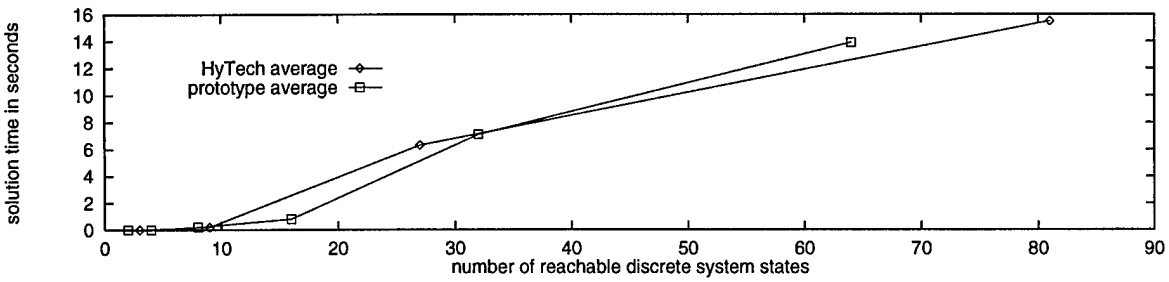


Figure 6: Average Solution Times as a Function of the Number of Discrete System States

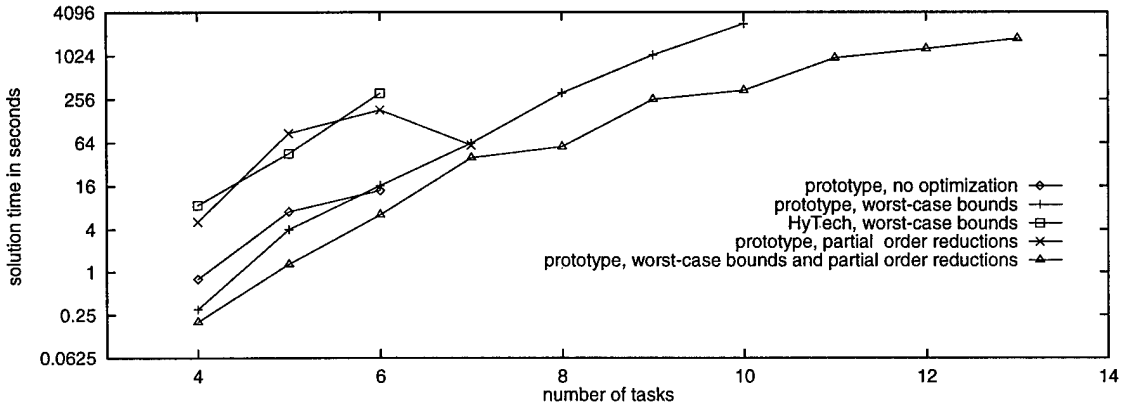


Figure 7: Prototype Average Solution Times with Optimization Methods

anyway.

By making scheduling a part of the model semantics rather than a part of the model itself, we reduced the size of the discrete state space from 3^t to 2^t where t is the number of tasks.[1] We cannot say with certainty how much of the performance improvement might be due to our different polyhedra operations versus the incorporation of scheduling semantics into the model. Figure 6 shows a plot of solution time versus number of reachable discrete system states rather than number of tasks, which seems to suggest the two different sets of polyhedra operations are about equally efficient. However, our prototype may be operating on more complex polyhedra, since information encoded in discrete states in HyTech must now be encoded in the polyhedra in our prototype. Also, with fewer discrete states, the set of polyhedra that must be checked for containment for each discrete state is larger for our prototype (the loop that searches for containing polyhedra accounts for most of the execution time). It may or may not be the case that some of HyTech's numeric problems occur when attempting to operate on polyhedra that are difficult for those algorithms in some fundamental way.

Classical uniprocessor schedulability analysis methods rely on the fact that analysis can be performed using only minimum periods and maximum compute times[19]. We simplified our models in this way. In our models there were frequently cases where periodic tasks were simultaneously dispatched. We added a test for these cases and performed such transitions concurrently, which is a partial order reduction method. Figure 7 compares the average solution times with and without these methods using a logarithmic scale. All solved problems were shown for all tools and methods at each point for which more than 75% of the problems were solved. The partial order method alone is actually slower than the unoptimized version for the task sets that could be analyzed in less than 1 hour, presumably because the time required to operate on polyhedra and the number of regions dominated the growth in the number of discrete system states to the extent that the additional testing required for simultaneous transitions (which requires among other things extra feasibility tests) is not worth the reduction in discrete state space size. As the figure suggests, partial order reduction did result in a somewhat higher percentage of problems being solved within one hour. The use of worst case values is significantly beneficial, this signifi-

[1] Although each task in the HyTech model had four discrete states, transitions to rescheduling states were forced to be simultaneous in many cases, and the size of the reachable discrete state space was only 3^t instead of 4^t.

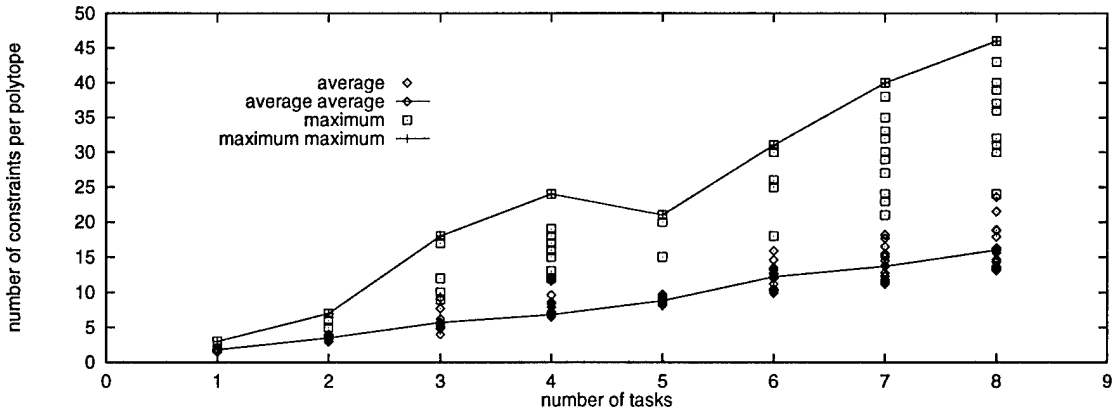


Figure 8: Average and Maximum Number of Constraints per Polyhedra

cantly reduces the number of enumerated regions. What this figure does not show is that this also significantly reduces the variability in number of regions and solution times. The figure shows that the two optimization methods are complementary and synergistic.

Figure 8 shows the average and maximum number of constraints required to represent polyhedra as a function of the number of tasks in a model. These experiments suggest that in practice the size of constraint sets grows roughly linearly with the size of models of this type.

7 Remarks

Our prototype tool and benchmarking exercises used only forward reachability analysis. It should be possible to perform a backward reachability analysis by using negated rates in the time successor operation, which would essentially run time backwards to obtain the time predecessor of a polyhedra[2, 3].

We attempted to generate problems that intuitively resembled real-world schedulability problems, but randomly generated problems may not be reflective of problems that would be encountered in practice. Moreover, the models we generated were precisely what would not be analyzed using linear hybrid automata techniques, since these models were amenable to analysis using traditional methods. Experience is needed with models that include features such as distributed execution, remote procedure calls, rendezvous, etc., the kinds of models for which we intended this analysis technique. Our preliminary experience with some simple distributed scheduling problems and with a software verification exercise suggests that our method works as well on these problems as on the uniprocessor benchmarks discussed in this paper[24].

We observed an increased ability to solve problems when only maximum compute times and minimum periods were used, which classical preemptive fixed priority theory tells us is sufficient for noninteracting tasks on uniprocessors. It is known that this is not true in certain multi-processor situations[10], and this is likely inadviseable when applying these methods to verification problems. It would be useful to identify more general conditions under which selected intervals in a model could be replaced by a scalar, or be expanded to a containing interval, in a way that significantly reduces the number of regions.

We added scheduling semantics to a linear hybrid automata model rather than explicitly modeling scheduling behavior using standard hybrid automata. The basic idea, which is to determine the variable rates (or edge guards or assignments) using a computable function of the composed system discrete states rather than a simple union of rates (or edge guards or assignments) specified in separate automata, might be applicable in other problem domains.

Our results suggest that the computational complexity of performing a reachability analysis does not lie in the complexity of the individual polyhedra operations but in the possible combinatorial explosion in the number of reachable regions, at least in practice in the problem domain we studied. Our work illustrates how the incorporation of domain semantics into the model and the use of a partial order reduction method can be effective in reducing the growth in the size of the discrete state space. An on-the-fly method might reduce the discrete state space size in some problem domains, but not in the one we studied. Several other researchers have explored methods that automatically approximate sets of polyhedra using a containing polyhedron to reduce the growth in the region space size[12, 16, 13], although we found during some preliminary experiments with some simple methods that it was difficult to simultaneously achieve acceptable accuracy and significant performance improvements. Our work illustrates another approach, conservative modifications to model parameters to better condition a model, that also holds some promise.

References

[1] K. Altisen, G. GöBler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, "A Framework for Scheduler Synthesis," *Real-Time Systems Symposium*, December 1999.

[2] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.

[3] Johan Bengtsson and Fredrik Larsson, *UPPAAL, A Tool for Automatic Verification of Real-Time Systems*, DoCS 96/97, Department of Computer Science, Uppsala University, January 15, 1996.

[4] B. A. Brandin and W. M. Wonham, "Supervisory Control of Timed Discrete-Event Systems," *IEEE Transitions on Automatic Control*, v39, n2, February 1994.

[5] A. L. Brearly, G. Mitra and H. P. Williams, "Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm," *Mathematical Programming*, 8, p54-83.

[6] David Bremner, personal communication, University of Washington Department of Mathematics, 1999.

[7] S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, "Computing Quantitative Characteristics of Finite-State Real-Time Systems," *Real-Time Systems Symposium*, December 1994.

[8] David L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," *International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 12-14, 1989, also in *Lecture Notes in Computer Science 407*, J. Sifakis (Ed.), Springer-Verlag, pp 197-212.

[9] Saul I. Gass, *Linear Programming*, McGraw-Hill Book Company, New York.

[10] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal of Applied Mathematics*, Vol. 17, No. 2, March 1969.

[11] Ronald L. Graham, Donald E. Knuth and Oren Patashnik, *Concrete Mathematics, A Foundation for Computer Science*, Addison-Wesley, 1989.

[12] Nicolas Halbwachs, Yann-Erik Proy and Patrick Roumanoff, "Verification of Real-Time Systems using Linear Relation Analysis," *Formal Methods in System Design*, 11(2):157-185, August 1997.

[13] Nicolas Halbwachs, Pascal Raymond and Yann-Eric Proy, "Verification of Linear Hybrid Systems by Means of Convex Approximations," *Workshop on Verification and Control of Hybrid Systems*, Piscataway, NJ, October 1995.

[14] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri and Pravin Varaiya, "What's Decideable About Hybrid Automata?" *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995.

[15] Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, "HyTech: The Next Generation," *Real-Time Systems Symposium*, December 1995.

[16] Thomas A. Henzinger and Pei-Hsin Ho, "A Note On Abstract Interpretation Strategies for Hybrid Automata," *Hybrid Systems II*, also *Lecture Notes in Computer Science 999*, Springer-Verlag, 1995.

[17] James P. Ignizio, Linear Programming in Single- and Multiple- Objective Systems, Prentice-Hall.

[18] J. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *IEEE Real-Time Systems Symposium*, 1989, pp 166-171.

[19] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, v20, n1, January 1973, pp 46-61.

[20] Peter J. G. Ramadge and W. Murray Wonham, "The Control of Discrete Event Systems," *Proceedings of the IEEE*, v77, n1, January 1989.

[21] Karsten Strehl, Lothar Thiele, Dirk Ziegenbein, Rolf Ernst and Jürgen Teich, "Scheduling Hardware/Software Systems Using Symbolic Techniques," *Proceedings of the 7^{th} International Workshop on Software/Hardware Codesign*, Rome, Italy, May 3-5, 1999, pp 173-177.

[22] Steve Vestal, "Fixed Priority Sensitivity Analysis for Linear Compute Time Models," *IEEE Transactions on Software Engineering*, April 1994.

[23] Steve Vestal, "Linear Hybrid Automata Models of Real-Time Scheduling and Allocation in Distributed Heterogeneous Systems," Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418, 1999.

[24] Steve Vestal, "Formal Verification of the MetaH Executive Using Linear Hybrid Automata," Honeywell Technology Center, Minneapolis, MN 55418, December 1999.

Formal Verification of the MetaH Executive Using Linear Hybrid Automata

Steve Vestal
steve.vestal@honeywell.com
Honeywell Technology Center
Minneapolis, MN 55418*

Abstract

MetaH is a language and toolset for the development of real-time high assurance software. There is an associated executive that is automatically configured by the tools to perform the task and message scheduling specified for an application. Linear hybrid automata are finite state automata augmented with real-valued variables. Transitions between discrete states may be conditional on the values of these variables and may reassign variables. These variables can be used to model real time and accumulated task compute time as well as program variables. We developed a concurrent linear hybrid automata model for that portion of the MetaH executive software that implements task scheduling and time partitioning. A reachability analysis was performed to verify selected properties for a selected set of application configurations. The approach combines aspects of testing and verification and automates much of the modeling and analysis. There are limits on the degree of assurance that can be provided, but the approach may be more thorough and less expensive than some traditional testing methods.

1 Introduction

Linear hybrid automata are finite state automata augmented with variables whose values change continuously in a way that depends on the current discrete state[4]. A variable's value may stay fixed in a given discrete state (have a rate or derivative of 0), or a variable's value may change continuously as time passes at a rate that may vary from discrete state to discrete state. Discrete transitions between discrete states may be conditional on the values of these variables and may reassign selected variables. Linear hybrid automata can be subjected to a reachability analysis to verify that a given set of assertions is true of a system. In general a semi-decision procedure must be used since the reachability problem for linear hybrid automata is undecidable[12], although it can be shown that reasonable pragmatic restrictions make fairly general models for real-time allocation and scheduling in distributed heterogeneous systems decidable[24].

Linear hybrid automata can be used to model many kinds of dynamical systems, but the problem of interest in this paper is the modeling of software that performs a real-time scheduling function. The continuous variables of a linear hybrid automaton can be used to model real time to specify deadlines, can be used as so-called integration variables to record accumulated compute time as tasks are preemptively scheduled, can be used to model hardware timers, and can be used to model program variables that appear in the software itself. Linear hybrid automata provide a concise and intuitive notation for describing complex real-time system structures and behaviors.

MetaH is an emerging SAE standard language for real-time fault-tolerant high assurance software architectures[2, 19, 20, 23]. Users specify how software and hardware components are combined to form an overall system. Our MetaH toolset can generate and analyze formal models for schedulability, reliability, and partition isolation. The toolset can also configure an application-specific executive to perform the specified task dispatching and scheduling, message and event passing, changes between alternative configurations, etc. Our executive supports a reasonably complex tasking model using preemptive fixed priority scheduling, including various forms of controlled time-slicing, error recovery, and time partitioning[5, 6, 7].

The core scheduling modules of the MetaH executive implement a set of discrete operations on tasks: start, stop, dispatch, complete, etc. These operations implement transitions between the discrete task scheduling states, e.g. dispatch may transition a task from the awaiting_dispatch state to the computing state. Continuously varying quantities, such as accumulating compute times and decreasing time slices, must also be modeled. We inserted calls to build a linear hybrid automata model of the executive code into the code itself. We developed several simple application specifications that included most (but not all) of the tasking features. We wrote a test driver that exercised all relevant paths in the core scheduling modules. The test driver thus triggered the generation of a linear hybrid automata model of all possible behaviors of the core scheduling operations for each application. We applied a reachability analysis algorithm to detect missed deadlines and a few other types of errors. Several defects were discovered, most involving incorrect handling of specific patterns of single and near-coincident multiple application-level faults, a few involving subtle timing defects such as race conditions.

Our primary new result is that this actually worked,

*This work has been supported by the Air Force Office of Scientific Research under contract F49620-97-C-0008.

and was moreover accomplished with a moderate and practical amount of effort. Linear hybrid automata reachability analysis is computationally and numerically a much more difficult problem than discrete state reachability analysis[13, 10], but using a recently developed method we were nevertheless able to analyze fairly detailed models of a piece of software of real-world complexity[25]. Our work suggests that this technology has reached the threshold of practical applicability, at least for the verification of small amounts of software of a particular type.

A secondary result is our verification method, which combines aspects of testing and formal methods. Our approach provides a high degree of traceability between code and model, automates certain aspects of model generation, and integrates reasonably well with the overall process and toolset.

We include a discussion of limitations on our results. We excluded many features from our initial modeling exercises, notably slack scheduling, interprocessor communication and dynamic reconfiguration. There are many potential defects of various types that our verification exercise could not have detected, and it is necessary to use this approach with other complementary verification methods. However, the effort required is relatively modest (perhaps comparable to unit testing), while the results seem more thorough and less expensive than what we believe would have been achieved using requirements testing of the verified features. We close with a discussion of possible future developments that may address some of the current limitations on applicability and thoroughness.

2 Linear Hybrid Automata

Figure 1 shows an example of a linear hybrid automaton. In addition to the discrete states `preempted`, `executing` and `waiting` there are also real-valued variables `c` and `t`. This example models a preemptively scheduled task, where `c` records the accumulated compute time (the total time spent in the `executing` state since the most recent dispatch transition) and `t` is a timer used to control periodic dispatching and to assert a deadline will be met. We need linear hybrid automata rather than the more tractable timed automata because we need integrator variables to model preemptive scheduling[3].

A discrete state may be annotated with an assignment of rates to the continuous variables of the automaton. The timer `t` in this example has rate 1 in all states, while `c` has rate 1 only when the task is executing and 0 otherwise. Discrete states may also be annotated with invariants to assure progress. In this example the invariant of state `executing` is that the value of `c` does not exceed 100, which models a task that never requires more than 100 units of compute time to complete.

Edges between discrete states may be guarded by a set of linear constraints over the real-valued variables. A transition can occur over an edge only when the variable values satisfy the guard. The guard `if c` $\geq$ 75 in this example models a task that computes for at least 75 units of time before completing (i.e. the task in this example may compute for a nondeterministic amount of time in the interval $[75, 100]$). Edges may also be labeled with assignments to real-valued variables, such as the resets of `c` and `t` on the edge that models a dispatch event for the example task.

Both discrete states and edges may be labeled with assertions, sets of linear constraints that should always be satisfied whenever the system is in the labeled discrete state and whenever a transition occurs over the labeled edge.

In this example the edge guards "selected" and "unselected" represent decisions made and actions taken by a task scheduler. The scheduler could be modeled as a concurrent control automaton[3]. It is also possible to encode preemptive fixed priority scheduling in a compositional hybrid automata task model using a shared queue variable together with scheduling guards on certain edges. However, we find it simpler and more general to add the desired scheduling semantics to a concurrent hybrid automata model and analyzer[24]. Our analyzer annotates individual task discrete states with resource and priority assignments and applies the desired scheduling semantics during the reachability analysis of a system composed from many tasks. Variable rates are computed by a scheduling function that looks at the states of all tasks in the composed system and returns the set of rates to use for that composed discrete state, very much analogous to the way run-time scheduling decisions are computed using a queue of ready tasks.

More formally, a state of a linear hybrid automaton consists of a discrete part, the discrete state at some time t; and a continuous part, the real values of the variables at time t. It turns out that, although this state space is uncountably infinite, the reachable state space for a given linear hybrid automaton is a subset of the cross-product of the discrete states with a recursively enumerable set of polyhedra in $\Re^n$ (where n is the number of variables)[4]. A region of a linear hybrid automaton is a pair consisting of a discrete state and a polyhedron, where polyhedra can be represented using a finite set of linear constraints. Model checking consists of enumerating the reachable regions for a given linear hybrid automaton and checking to see if they satisfy the assertions. There is no guarantee that this enumeration will terminate, but if it does then assertions have been verified in all possible regions reachable by the hybrid automaton.

Figure 2 depicts the basic sequence of operations that, given a starting region (a discrete state and a polyhedron defining a set of possible values for the variables), computes the set of values the variables might take on in that discrete state as time passes; and computes a set of regions reachable by subsequent discrete transitions.

The first step is the computation of the time successor polyhedron from the starting polyhedron (often called the post operation). For each point in the starting polyhedron, the time successor of that point is a line segment beginning at that point whose slope is defined by the variable rates specified for the discrete state. This is the set of variable values that can be reached from a starting point by allowing some amount of time to pass. The time successor of the starting polyhedron is the union of the time successor lines for all points in the starting polyhedron. A

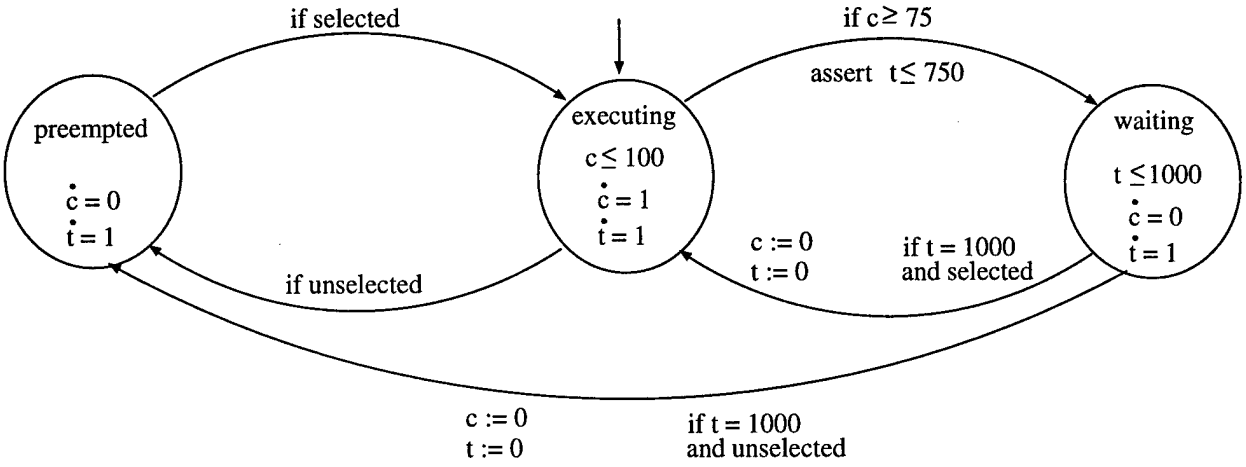


Figure 1: A Hybrid Automata Model of a Preemptively Scheduled Task

basic result of linear hybrid automata theory is that the time successor of any polyhedron is itself a polyhedron (which in general will be unbounded in certain directions)[4].

The second step is the intersection of the time successor polyhedron with the invariant constraint associated with the discrete state, yielding the time successor region that is feasible given the specified invariant. Polyhedra are easily intersected by taking the union of the set of linear constraints that define the two polyhedra.

The remaining steps are used to compute new regions reachable from this feasible time successor region by some transition over an edge. For each edge out of the current discrete state, the associated guard is first intersected with the feasible time successor region. This polyhedron, if nonempty, defines the set of all variable values that might exist whenever the discrete transition could occur. Any variable assignments associated with the edge must now be applied to this polyhedron. This is done in two phases. First, a variable to be assigned a new value $x := l$ is unconstrained (often called the free operation). This operation leaves unchanged the relationships between all other variables, i.e. the polyhedron is projected onto the subspace $\Re^{n-1}$ of the remaining variables. This result is then intersected with the constraint $x = l$. This polyhedron, together with the discrete state to which the edge goes, is a new region for which the above steps may be repeated. In general a set of assignments whose right-hand sides are linear formula are allowed, with some restrictions. The variables to be assigned are unconstrained and the resulting polyhedra are then intersected with the appropriate linear constraints in some order. As we will see, this allows us to model fairly complex sequences of assignments to program variables.

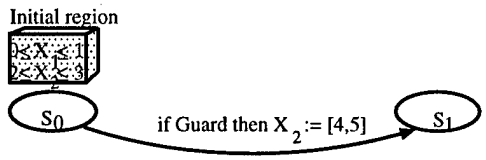
Analysis begins at the initial region of a hybrid automaton. The operations described above are applied to enumerate feasible time successor regions and the new regions reachable from these via discrete transitions. As new regions are enumerated, they must be checked to see if they have been visited before (otherwise the method will not terminate even when there are a finite number of regions). This is done by comparing the discrete states of regions for equality, and by checking to see if the new polyhedron is contained in a previously enumerated polyhedron for that discrete state.

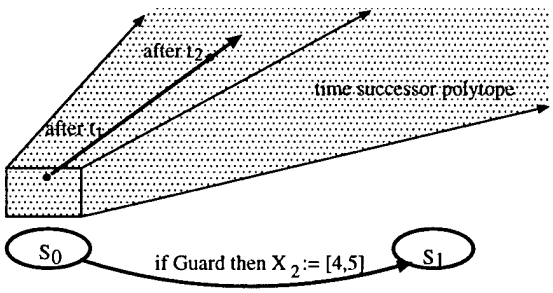
There are previously developed tools that perform a reachability analysis for linear hybrid automata[13, 10]. The work described in this paper uses new reachability analysis methods we developed that enabled us to analyze larger problems[25]. Our tool incorporates scheduling semantics and allows sequences of assignments on edges.

3 MetaH

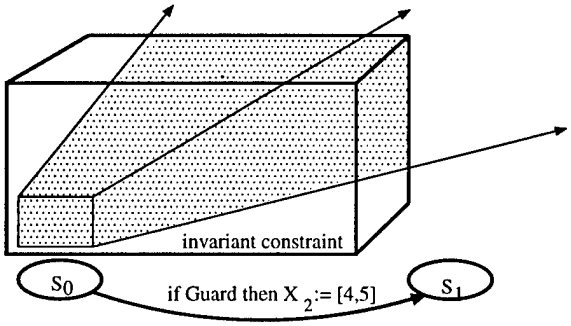
MetaH is a language for specifying software and hardware architectures for real-time, time-and-space partitioned, fault-tolerant, scalable multi-processor systems. Developers specify how a system is composed from software components like tasks and packages and hardware components like processors and memories. An associated toolset performs syntactic and semantic checks, generates and analyzes various models of the system, and automatically tailors a system executive to integrate all the components together as specified. Figure 3 shows the current toolset.

Low-level software constructs of the MetaH language describe source components written in a traditional programming language (currently Ada, C and C++). The source components themselves are developed using other methods and specialized tools, e.g. reengineered, autocoded[19, 20]. Subprogram and package specifications describe the aspects of source modules that are needed to integrate them into a system, e.g. name of the file containing the source code, nominal and maximum compute times on various kinds of processors, stack and heap requirements. Events (user-specified enumeration literals used in certain service calls) and ports (buffer variables used to hold message values) can appear within source modules and must be accurately specified (the tools will

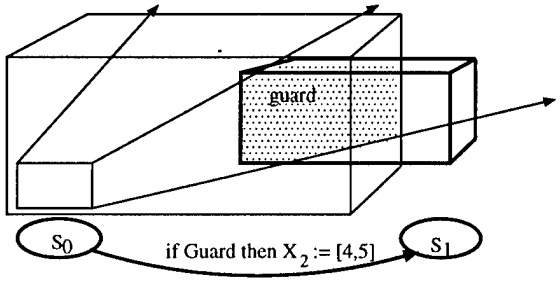




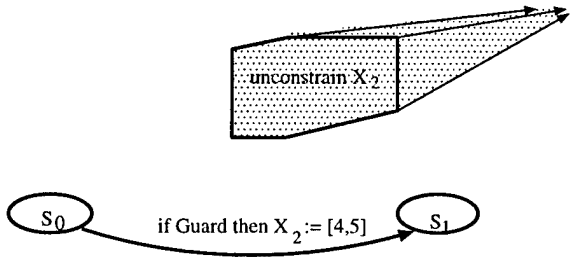
Step 1: Time Successor



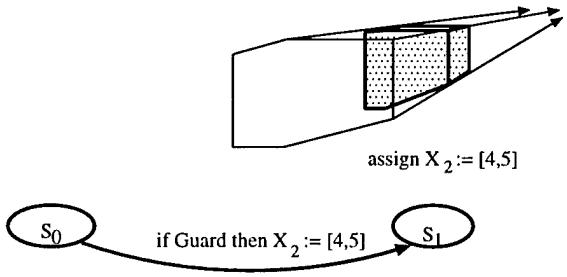
Step 2: Intersection with Invariant Constraint



Step 3: Intersection with Transition Guard



Step 4: Unconstrain Assigned Variable



Step 5: Intersect with Variable Assignment

Figure 2: Hybrid Automata Reach Forward Operations

parse selected source files and check for consistency with the specifications).

The higher-level software constructs of the MetaH language are processes, macros and modes. Processes group together source modules that are to be scheduled as either periodic or aperiodic tasks. A process is also the basic unit of fault containment, and memory protection and compute time enforcement are supported for some target processor configurations. Macros and modes group processes, define connections between events and ports, and define equivalences between packages that are to be shared between processes. The difference between macros and modes is that macros run in parallel with each other, while modes are mutually exclusive and are used to specify alternative run-time configurations. Event connections between modes are used to define hierarchical mode transition diagrams, where mode changes at run-time can stop or start processes or change connections.

MetaH allows hardware architectures to be specified using memory, processor, channel, and device components grouped into systems. Hardware objects may have ports, events or packages in their interfaces. Software and hardware ports and events can be connected to each other, and software can access hard-

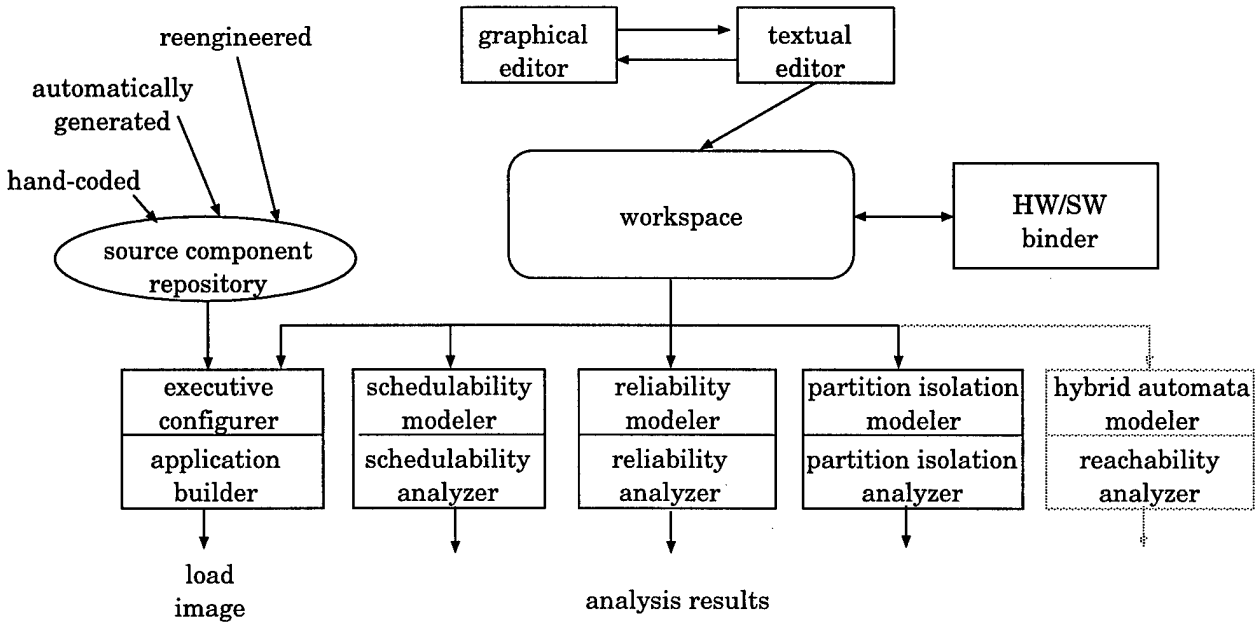


Figure 3: MetaH Toolset

ware packages (hardware packages provide hardware-dependent service calls). Hardware descriptions identify (among other things) hardware-dependent source code modules for device drivers, and code to provide a standard interface between the automatically configured executive and the underlying real-time operating system.

A simple software/hardware binding tool assigns to hardware those software objects in a specification that are not explicitly assigned, possibly subject to user-specified constraints.

Using information contained in the MetaH specification and produced by the executive configuration tool, the schedulability modeler generates a detailed preemptive fixed priority schedulability model of the application. The model includes all scheduling and communication overheads. The schedulability analysis algorithm we currently use is an extension of the exact characterization algorithm[18, 22]. Our analysis tool produces sensitivity analysis information describing how compute times may be changed while preserving (or in order to achieve) schedule feasibility; and it allows processes to be decomposed into component source modules and provides analysis data for individual source modules.

The MetaH language includes a construct called an error model, which allows users to specify sets of fault events and error states and behaviors. Objects within a specification can be annotated to specify the error model, specific fault rates to use for that object, and consensus expressions to model voting protocols and fault-tolerance requirements. We have a prototype reliability modeling tool that generates a stochastic concurrent process reliability model[16, 21]. A subset of the reachable state space of this stochastic concurrent process is a Markov chain that can be analyzed using existing tools and techniques[16, 17].

MetaH can support time and space partitioning[1]. Protected address spaces and special scheduling techniques are used in a way that allows certain guarantees to be made about fault containment. If it can be shown that a defect in one component cannot possibly cause incorrect behavior in another component, then it is possible to simplify or eliminate certain verification activities. The partition isolation modeling and analysis tool checks such conditions. We note that the MetaH executive, which enforces partitioning and was the object of our formal verification work, must be verified to the highest level.

4 MetaH Executive Modeling

Figure 4 shows the high-level structure of the MetaH executive. The core task scheduling operations are implemented by `Threads`, e.g. start, dispatch, complete. `Threads` invokes operations in `Time_Slice`, which encapsulates arithmetic operations and tests on two execution time accumulators maintained by the underlying RTOS and hardware for each task: an accumulator that increases while a task executes, and a time slice that decreases while a task executes. `Time_Slice` may set these variables to desired values using services provided through the `MetaH RTOS interface`. If time slicing is enabled for a task, then a trap will be raised by the underlying hardware and RTOS when the time slice reaches zero. This trap is handled by one of the operations in `Threads`. `Clock_Handler` is periodically invoked by the underlying system (it is the handler for a periodic clock interrupt) and makes calls to `Threads` to dispatch periodic tasks and start and stop threads at mode changes. `Events`, `Modes` and `Semaphores` contain data tables and operations to manage user-declared events, dy-

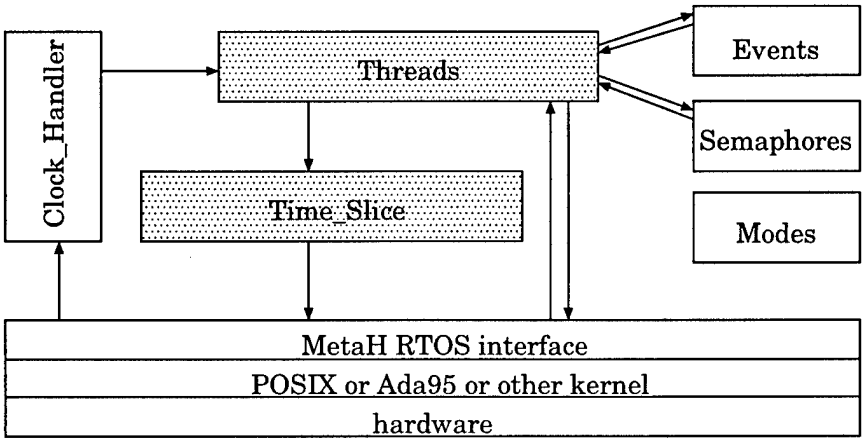


Figure 4: MetaH Executive Structure

namic reconfiguration, and semaphores.

Most of these modules include code that is automatically generated by the configuration tool and code that is common across all applications. `Threads`, `Time_Slice` and `Semaphores` use only generated data tables, all the executable code is common to all applications. `Modes` contains only generated data tables and no executable code at all. `Clock_Handler` includes some automatically generated case statements that periodically call `Threads` operations at different rates in different modes. `Events` includes some automatically generated case statements to vector events to the proper `Threads` operations in different modes.

As Figure 4 illustrates, `Threads` provides the core scheduling operations that are invoked by the other modules. Only `Threads` and `Time_Slice` make calls to underlying RTOS services such as suspend, resume, and set priority. All trap and interrupt events eventually result in calls to `Threads` operations. `Threads` and `Time_Slice` account for about 1800 of the 2800 lines of non-application-specific, non-target-specific code in the executive. It is these two modules for which we created linear hybrid automata models.

In essence, each `Threads` operation is a kind of event that transitions a thread from one scheduling state to another. `Threads` implements 13 basic scheduling operations on tasks. Figure 5 illustrates the general code structure for an operation. There is a global array that maintains state information about each task, including the scheduling state (there are 15 scheduling states). For example, the `Dispatch` operation in Figure 5 will transition a thread in the `Awaiting_Dispatch` state to the `Computing` state.

Figure 5 also illustrates how the operation can be influenced by other factors, such as the application developer's choice for the handling of dispatch events that arrive for an aperiodic process while it is still servicing a previous event. Many decisions made during execution are dependent on the task subtype (periodic, transformed periodic, incremental periodic, period enforced aperiodic, slack scheduled aperiodic). Much of the data used to make decisions in the code is contained in declarations and tables generated by the MetaH toolset. In particular, an array of static data is generated that describes for each task its subtype, whether it has enforced execution time limits and what they are, its scheduling priority, the semaphores it is allowed to lock and unlock, etc.

The implementation is complicated by the fact that the task state transition diagram is hierarchical rather than flat. The scheduling states associated with semaphore locking and unlocking, and with the controlled time-slicing of period transformed and slack scheduled tasks, form sub-state-machines that can be called, subprogram-like, from any of the starting or computing or recovering states. A state variable (`Return_To_State` in Figure 5) is used in the code to record the return state. (The current executive implementation does not supported nested semaphore locking.)

Rather than write a separate model of this code using a specialized modeling language, we inserted model-building calls into the `Threads` and `Time_Slice` code. For example, the code in Figure 5 had a call inserted to a subprogram that would create an edge between the `Await_Dispatch` and the `Computing` modeling states for the task identified by the `Thread` parameter. A test driver was written that first created discrete model states for all tasks in the application, then called each `Threads` operation. Subprograms were added to `Threads` so that this test driver could first explicitly set the values of any relevant internal state variables. The test driver called every operation on every thread with every possible setting of modeled internal state variables. To put this another way, the test driver exercised every modeled path through the `Threads` and `Time_Slice` code, thereby including in the model every possible modeled transition that might occur at run-time for every task.

A special target processor specification was developed that provided a null MetaH processor interface, except that the test driver was substituted for the normal code to begin application execution. From the user's perspective, a new target processor specification was added to the MetaH library, where the execution of an application built for this processor re-

```
Thread_State_Type is
    record
        Scheduling_State: State_Type;
        Dispatch_Enqueued: Boolean;
        Return_To_State: State_Type;
    end record;

State: array (Thread_ID_Type) of Thread_State_Type;

procedure Dispatch (Thread: Thread_ID_Type;
                    Busy_Option: Option_Type) is
begin
   case State(Thread).Scheduling_State is
   when Awaiting_Dispatch =>
      State(Thread).Scheduling_State := Computing;
      Time_Slice.Dispatch (Thread);
      MetaH_Processor_Interface.Resume (Thread);
   when Computing =>
      case Busy_Option is
      when Ignore_Dispatch =>
         null;
      when Enqueue_Dispatch =>
         State(Thread).Enqueue_Dispatch := True;
      when Raise_Fault =>
         Threads.Thread_Error (Thread, Lost_Event);
      end case;
   when ... =>
   end case;
end Dispatch;
```

Figure 5: Thread Code Structure

sulted in the generation and analysis of a linear hybrid automata model for that application (after a few special modeling fields added to the task data tables were filled in by hand). The modeling and analysis for any specific application specification is thus largely automated and appears as a new analysis tool as illustrated in Figure 3.

The timing of the calls to some Threads operations is important. Periodic tasks are dispatched at a specified rate by the Clock_Handler, a module that we did not model explicitly. The behavior (but not the detailed code structure) of Clock_Handler is implicitly modeled using timer variables for each task, where a guard on each dispatch model edge enables a dispatch transition only when the timer equals the specified task period, and an assignment on each dispatch model edge then resets that timer to zero (recall the use of variable t in Figure 1). Timer variables are also used to model the timing of certain period transformation and period enforcement operations.

The models specified that starting, computing, recovering and locking times might be anywhere between zero and the nominal values in the specification. For systems with enforced execution times, we specified nominal compute values that would cause deadlines to be missed if any limit was improperly enforced. The nominal compute times exceeded the enforced maximum compute times in these specifications (not what one would normally encounter), and the toolset was directed to use enforced rather than nominal compute times for schedulability analysis. For aperiodic tasks the model allowed the period between successive dispatch events to be any value between half the enforced period and twice the system hyperperiod.

We also introduced model variables for certain program variables that appear in the implementation code. For example, the variable that records whether an event is enqueued for an aperiodic task was modeled (this is a bounded queue, currently bounded at a single event). Guards involving this variable determine whether a busy aperiodic task transitions to the Awaiting_Dispatch, Computing or Recovering state when it becomes possible to service another event. The variable that records the Return_To_State for the semaphore locking and period transformation sub-state-machines was modeled. This variable is assigned on every edge that enters one of these sub-state-machines. Edges appear in the model from these substates back to every possible calling state, where a guard determines which state is actually transitioned to based on the value of the Return_To_State variable.

Finally, we introduced variables to model controlled time slicing and execution time enforcement. This was the most complicated aspect of our modeling work, partly because there is some nested program logic split between Threads and Time_Slice, and partly because there are some subtle issues involved in modeling the complex sequences of assignments that appear in the code. Even then we simplified our problem by not modeling slack scheduling and excluding slack-scheduled task types in our analyses.

In order to enforce blocking time bounds, the MetaH executive can enforce semaphore locking time limits that are distinct from overall compute time limits. One of the most complex time variable manipulations involves the saving and restoring of execution time information when semaphores are locked and unlocked. The exact set of assignments performed by the lock and unlock operations is determined by program logic (modeled using multiple edges, guards and assignments), but one sequence of assignments that is sometimes executed at a lock is

```
Previous_Time_Slice(T) := Time_Slice(T);
Previous_Execution_Time(T) := Execution_Time(T);
Time_Slice(T) := Locking_Limit (T, S);
```

followed at the unlock by

```
Time_Slice(T) := Previous_Time_Slice(T)
              - (Execution_Time(T)
                 - Previous_Execution_Time(T));
```

Time_Slice appears on both the right-hand and the left-hand side of the first block of assignments. These assignments must be modeled in multiple phases: free Previous_Time_Slice and Previous_Execution_Time; intersect the constraints Previous_Time_Slice = Time_Slice and Previous_Execution_Time = Execution_Time; and then a final assignment (free then constrain) to Time_Slice. It was not necessary to introduce intermediate discrete states to model a sequence of assignments, but it was necessary to extend the reachability tool to allow sequences of assignments on a transition edge.

A second and more subtle effect is due to the semantics of time succession. `Time_Slice` and `Execution_Time` are not normal program variables, they are time-varying accumulators maintained by the underlying RTOS using hardware timers. Whenever task T is executing, the values of these variables change at rates of -1 and 1 respectively. In the unlock assignment, if we simply imposed the constraint

```
Time_Slice(T) = Previous_Time_Slice(T)
              - (Execution_Time(T)
                 - Previous_Execution_Time(T));
```

then during the next time successor operation at which task T is executing, both `Time_Slice` and `Execution_Time` would change. The time successor operation will maintain this constraint between the two time-varying variables as time passes. However, what we really want to model is that the value of `Time_Slice` depends on what the value of `Execution_Time` was at a particular instant of time, the most recent unlock of the semaphore. In order to do this we must introduce a zero-rate sampling variable and model the assignment as if it had been coded

```
Sampled_Execution_Time(T) := Execution_Time(T);
Time_Slice(T) := Previous_Time_Slice(T)
              - (Sampled_Execution_Time(T)
                 - Previous_Execution_Time(T));
```

The assignment constraint imposed at the time of this transition will constrain `Time_Slice` relative to zero-rate variables only as time passes. The value of `Time_Slice` as time passes will depend on the sampled execution time when the unlock occurred rather than on the time-varying `Execution_Time` accumulator.

A total of 14 real-valued variables and 15 discrete states were defined to model each task. No single task model used all 14 variables and 15 states, different task types with different specified options used different combinations. In order to minimize the number of variables that had to be manipulated during reachability analysis, we inserted free operations for selected variables on selected edges to undefine and remove them from the model (e.g. `Return_To_State` was freed on each edge leading out of one of the sub-state-machines). Figure 6 shows the simplest linear hybrid automata model we generated, a periodic task with period and deadline of 100000us, compute time between 0 and 90000us, recovery time between 0 and 10000us. States are also annotated with processor scheduling priorities, which are not shown here. The variable rates were derived from the scheduling priorities by the analysis tool, which used preemptive fixed priority scheduling semantics for this study.

The conditions we checked during reachability analysis were that all deadlines were met whenever the schedulability analyzer said an application was schedulable; no accessed variables were undefined and no invariants were violated on entry to a region; and no two tasks were ever in a semaphore locking state simultaneously. In addition, the code itself includes some assertion checks. These were modeled by edges annotated with `assert False`.

Current reachability analysis methods are not capable of analyzing applications of real-world complexity involving several tasks. However, the modeled code does not change from application to application, only the configuration tables that it uses. Our goal was to obtain some assurance that this fixed code is correct using a specific finite set of verification exercises. We developed a set of MetaH application specifications for one and two task systems that modeled a variety of combinations of task subtypes and tasking options, described in Table 1. Our results to date are a hybrid between testing and verification, where we verify the code for a specific set of configuration tables for a specific set of applications. Each exercise verifies correctness for all possible compute times and event sequences that are possible in each application configuration.

We defined a verification coverage metric to increase our assurance of correctness. We collected information about which edges were used by some transition during reachability analysis and compared this with all the possible edges that might be created (all instances of calls inserted into the code to create edges). This metric determined if a piece of code was modeled by an edge that appears in the model and also insured that some analysis examined possible run-time transitions over that edge, analogous to measuring path coverage during unit testing. Edges used to model internal code assertion failures caused no run-time transitions. A few edges were not used because we omitted dynamic reconfiguration from our exercise, one edge would have been used only under overload conditions. After examining this data for our initial set of applications, we discovered that two additional configurations were needed to achieve complete coverage of the remaining edges, one of which revealed a defect that had gone undetected.

5 Results

We discovered nine defects in the course of our verification exercise. Four of these were tool defects, two that could cause bad configuration data to be generated and two that could cause erroneously optimistic schedulability models to be generated. Six of these defects could cause errors only during the handling of application faults and recoveries, three of these six only in the presence of multiple near-coincident faults and recoveries.

We have not performed a systematic unit testing of the threads operations, but we believe the effort required for our modeling and analysis is roughly comparable. The effort required to develop test drivers and test cases that will drive execution along all paths is probably comparable. Unit testing does not require the insertion of modeling code, but does sometimes require the development of code to set internal variables and observe intermediate results. Also, we did not have to determine acceptable results for each unit test.

We have not performed a systematic requirements testing of the MetaH toolset and executive, but we believe the approach outlined in this paper provides more thorough defect detection than is likely to be achieved in practice using requirements testing of the features we modeled. The thoroughness of requirements testing depends partly on the available resources and

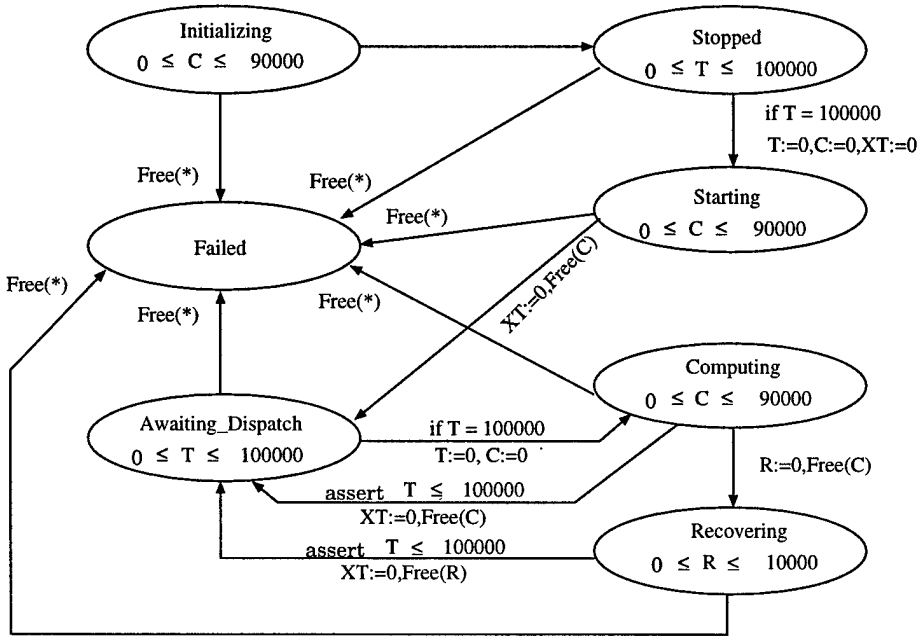


Figure 6: Generated Hybrid Automata Model for a Simple Periodic Task

Description	Discrete States	Distinct Polyhedra	Sparc Ultra-2 CPU Seconds
one periodic task	7	7	0
one periodic task, enforced execution time limits	7	10	0
one periodic task, enforced execution time limits, one semaphore	8	29	15
one period-enforced aperiodic task	9	18	0
one period-enforced aperiodic task, enforced execution time limits	9	27	2
one period-enforced aperiodic task, enforced execution time limits, one semaphore	11	124	125
two periodic tasks	36	60	3
two periodic tasks, enforced execution time limits	36	108	24
two periodic tasks, one with period transformed into two pieces,	41	97	10
two periodic tasks, one shared semaphore	48	118	36
two periodic tasks, one with period transformed into two pieces, enforced execution time limits	41	174	87
two periodic tasks, one with period transformed into four pieces, enforced execution time limits, recovery limit greater than compute limit	40	334	103
two tasks, one periodic and one period-enforced aperiodic	44	623	115
two periodic tasks, one with period transformed into four pieces, enforced execution time limits	41	351	170
two tasks, one periodic and one period-enforced aperiodic, enforced execution time limits	44	425	184
two tasks, one periodic and one period-enforced aperiodic, one shared semaphore	70	638	840
two periodic tasks, one with period transformed into two pieces, enforced execution time limits, one shared semaphore	55	963	5658

Table 1: Modeled Applications

partly on the experience and perversity of the individuals doing the testing. In our judgement, of the nine defects we found, one would almost certainly have been detected by moderately thorough requirements testing, while three would have been almost impossible to detect by testing due to the multiple carefully timed events required to produce erroneous behavior. The other five may have been detected by thorough requirements testing of fault and recovery features, providing the tester thought about possible execution timelines and arranged for tasks to consume carefully selected amounts of time between events.

Structured review (inspection, walk-through) is another common and highly effective method for detecting defects. In some ways our modeling effort resembled a semi-formal review of the design and code. The generated linear hybrid automata models, which we printed in a readable form, provided a good abstract description of the scheduling behavior. This description had good intuitive traceability both to the code and to some of the original requirements specifications. We recommend that the approach outlined in this paper be broadly construed to include explicit review of the design, code and linear hybrid automata models. One of the defects we detected was discovered during a review of the linear hybrid automata model rather than by reachability analysis.

We did not explicitly model the details of `Clock_Handler` or `Events`, which are generated application-specific modules. We modeled the mode change (dynamic reconfiguration) that occurs at system start-up but not any application-triggered mode changes. Our models did not account for the fact that executive operations require a small but finite amount of time and are performed non-preemptively. We did not model any multi-processor behaviors. These can all be modeled in a fairly straight-forward and intuitive way using linear hybrid automata, but we cannot say at what point the models would become unanalyzable using our current reachability methods. Whether or not slack scheduling or complex priority inheritance protocols could be easily modeled is an open question.

There are many non-scheduling executive behaviors that were not modeled, including some behaviors supported in part by code in `Threads` such as message copying, instrumentation, and status reporting. In our judgement, it is fairly easy to informally distinguish between specified behaviors that are modeled, and specified behaviors that are not modeled and would need to be verified using traditional requirements testing or other methods.

A number of limitations would exist even with the most detailed and complete linear hybrid automaton model. Some sort of induction argument is needed to establish correctness of the scheduling code for all possible application configurations. The MetaH processor interface, underlying RTOS and hardware were not modeled and are unlikely to be fully model-able for a variety of practical and technical reasons. The MetaH tools were not verified, only a few specific generated modules and reports for a few example applications. Although our approach provides good traceability between code and model, there is still a very real possibility of modeling errors. The reachability analysis tool may contain defects; we discovered two in our tool in the course of this work. Even if the source code is correct, defects in the compiler, linker or loader software could introduce defects into the executable image.

6 Future Work

It should be possible to use the set of reachable regions produced by the analysis tool to automatically generate tests. This could significantly reduce the cost and increase the quality of requirements testing (which might still be required by the powers-that-be). Such tests could also detect defects that could not be found by model analysis, such as defects in the compiler, linker, loader, RTOS or hardware. One of the issues that must be confronted is the ease of constructing, running and observing the results of tests; for example, in theory one might encounter transitions in the model that occur only when an accumulated execution time is exactly equal to some fixed value, which would be practically impossible to do in a test. Another issue is that such tests would not take into account the internal logic of unmodeled modules such as the RTOS; a systematic method for testing multiple points within each reachable polyhedron might help address this.

There are a number of potentially useful improvements in analysis methods and tools. Approximation and partial order methods might significantly increase the size of the model that could be analyzed[10, 14, 11, 25]. It is possible to apply theorem proving methods to linear hybrid automata[15], and some work has been done on dense-time process algebras[8, 9]. Decomposition and induction methods currently being explored for discrete state models might be extensible to linear hybrid automata. There are a number of possible ways to visualize and navigate the reachable region space that would be of practical assistance during model development and debugging and during reviews. Concise APIs and support for in-line modeling could reduce both the modeling effort and the number of modeling defects.

Changes will inevitably be required to the design, implementation and verification processes to make good use of these methods. An important and not completely technical question is how verification processes might be changed to beneficially use these methods. What evidence would be required, for example, to convince a development organization or regulatory authority to replace selected existing verification activities with modeling and analysis activities, or to add modeling and analysis to current verification activities?

References

[1] *Design Guidance for Integrated Modular Avionics*, AEEC/ARINC 651, Airlines Electronic Engineering Committee/ Aeronautical Radio Inc., 1991.

[2] *MetaH User's Guide*, Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN, `www.htc.honeywell.com/metah`.

[3] K. Altisen, G. Göbler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, "A Framework for Sched-

uler Synthesis," *Real-Time Systems Symposium*, December 1999.

[4] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, March 1996, pp 181-201.

[5] Pam Binns, "Scheduling Slack in MetaH," *Real-Time Systems Symposium*, work-in-progress session, December 1996.

[6] Pam Binns, "Incremental Rate Monotonic Scheduling for Improved Control System Performance," *Real-Time Applications Symposium*, 1997.

[7] Pam Binns and Steve Vestal, "Message Passing in MetaH using Precedence-Constrained Multi-Criticality Preemptive Fixed Priority Scheduling," Honeywell Technology Center, Minneapolis, MN 1999.

[8] Patrice Brémond-Grégoire and Insup Lee, "A Process Algebra of Communicating Shared Resources with Dense Time and Priorities," University of Pennsylvania Department of Computer Science Technical Report MS-CIS-95-08, June 1996.

[9] Andre N. Fredette, *A Generalized Approach to the Analysis of Real-Time Computer Systems*, Ph.D. Dissertation, North Carolina State University, March 1993.

[10] Nicolas Halbwachs, Yann-Erik Proy and Patrick Roumanoff, "Verification of Real-Time Systems using Linear Relation Analysis," *Formal Methods in System Design*, 11(2):157-185, August 1997.

[11] Nicolas Halbwachs, Pascal Raymond and Yann-Eric Proy, "Verification of Linear Hybrid Systems by Means of Convex Approximations," *Workshop on Verification and Control of Hybrid Systems*, Piscataway, NJ, October 1995.

[12] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri and Pravin Varaiya, "What's Decideable About Hybrid Automata?" *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, 1995.

[13] Thomas A. Henzinger, Pei-Hsin Ho and Howard Wong-Toi, "HyTech: The Next Generation," *Real-Time Systems Symposium*, December, 1995.

[14] Thomas A. Henzinger and Pei-Hsin Ho, "A Note On Abstract Interpretation Strategies for Hybrid Automata," *Hybrid Systems II*, also *Lecture Notes in Computer Science 999*, Springer-Verlag, 1995.

[15] Thomas A. Henzinger and Vlad Rusu, "Reachability Verification for Hybrid Automata," *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control*, also *Lecture Notes in Computer 1386*, Springer-Verlag, 1998.

[16] Holger Hermanns, Ulrich Herzog and Vassilis Mertsiotakis, "Stochastic Process Algebras as a Tool for Performance and Dependability Modeling," *Proceedings of the IEEE International Computer Performance and Dependability Symposium* (IPDS'95), April 24-26, 1995, Erlangen, Germany.

[17] Allen M. Johnson, Jr. and Miroslaw Malek, "Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability," *ACM Computing Surveys*, v20 n4, December 1988.

[18] J. Lehoczky, L. Sha and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *IEEE Real-Time Systems Symposium*, 1989, pp 166-171.

[19] Bruce Lewis, "Software Portability Gains Realized with MetaH, an Avionics Architecture Description Language," 18th *Digital Avionics Systems Conference*, St. Louis, MO, October 24-29, 1999.

[20] David J. McConnell, Bruce Lewis and Lisa Gray, "Reengineering a Single Threaded Embedded Missile Application onto a Parallel Processing Platform using MetaH," *Proceedings of the 4th Workshop on Parallel and Distributed Real-Time Systems*, 1996.

[21] Frederick T. Sheldon, Krishna M. Kavi and Farhad A. Kamangar, "Reliability Analysis of CSP Specifications: A New Method Using Petri Nets," *Proceedings of AIAA Computing In Aerospace*, San Antonio, TX, March 28-30, 1995.

[22] Steve Vestal, "Fixed Priority Sensitivity Analysis for Linear Compute Time Models," *IEEE Transactions on Software Engineering*, April 1994.

[23] Steve Vestal, "An Architectural Approach for Integrating Real-Time Systems," *Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1997.

[24] Steve Vestal, "Linear Hybrid Automata Models of Real-Time Scheduling and Allocation in Distributed Heterogeneous Systems," Honeywell Technology Center, Minneapolis, MN 1998.

[25] Steve Vestal, "A New Linear Hybrid Automata Reachability Procedure," Honeywell Technology Center, Minneapolis, MN 1999.