Report Number: WUCSE-2007-34

2007

Scheduling Induced Bounds and the Verification of Preemptive Real-Time Systems

Terry Tidwell, Christopher Gill, and Venkita Subramonian

Distributed real-time and embedded (DRE) systems have stringent constraints on timeliness and other properties whose assurance is crucial to correct system behavior. Our previous research has shown that detailed models of essential middleware mechanisms can be developed, composed, and for constrained examples verified tractably, using state of the art timed automata model checkers. However, to apply model checking to a wider range of real-time systems, particularly those involving more general forms of preemptive concurrency, new techniques are needed to address decidability and tractability concerns. This paper makes three contributions to research on formal verification and validation of DRE systems.... **Read complete abstract on page 2.**

Scheduling Induced Bounds and the Verification of Preemptive Real-Time Systems

Terry Tidwell, Christopher Gill, and Venkita Subramonian

Complete Abstract:

Distributed real-time and embedded (DRE) systems have stringent constraints on timeliness and other properties whose assurance is crucial to correct system behavior. Our previous research has shown that detailed models of essential middleware mechanisms can be developed, composed, and for constrained examples verified tractably, using state of the art timed automata model checkers. However, to apply model checking to a wider range of real-time systems, particularly those involving more general forms of preemptive concurrency, new techniques are needed to address decidability and tractability concerns. This paper makes three contributions to research on formal verification and validation of DRE systems. First, it describes how bounded fair scheduling policies introduce a quasi-cyclic structure in the state space of multi-threaded real-time systems. Second, it shows that bounds on the divergence of threads' execution can be determined for that quasi-cyclic structure, which then can be exploited to reduce the complexity of model checking. Third, it presents a case study involving progress-based fair scheduling of multi-threaded processing pipelines, with which the approach is evaluated.

2007-34

Scheduling Induced Bounds and the Verification of Preemptive Real-Time Systems

Authors: Tidwell, Terry; Gill, Christopher; Subramonian, Venkita;

Corresponding Author: cdgill@cse.wustl.edu

Abstract: Distributed real-time and embedded (DRE) systems have stringent constraints on timeliness and other properties whose assurance is crucial to correct system behavior. Our previous research has shown that detailed models of essential middleware mechanisms can be developed, composed, and for constrained examples verified tractably, using state of the art timed automata model checkers. However, to apply model checking to a wider range of real-time systems, particularly those involving more general forms of preemptive concurrency, new techniques are needed to address decidability and tractability concerns. This paper makes three contributions to research on formal verification and validation of DRE systems. First, it describes how bounded fair scheduling policies introduce a quasi-cyclic structure in the state space of multi-threaded real-time systems. Second, it shows that bounds on the divergence of threads' execution can be determined for that quasi-cyclic structure, which then can be exploited to reduce the complexity of model checking. Third, it presents a case study involving progress-based fair scheduling of multi-threaded processing pipelines, with which the approach is evaluated.

Notes:
Research at Washington University was supported in part by NSF awards CCF-0615341 (EHS) and

Type of Report: Other

Scheduling Induced Bounds and the Verification of Preemptive Real-Time Systems *

Terry Tidwell and Christopher Gill
CSE Department, Washington University
St. Louis, MO, USA
{ttidwell,cdgill}@cse.wustl.edu

Venkita Subramonian
AT&T Labs, Inc.
Florham Park, NJ, USA
venkita@research.att.com

ABSTRACT

Distributed real-time and embedded (DRE) systems have stringent constraints on timeliness and other properties whose assurance is crucial to correct system behavior. Our previous research has shown that detailed models of essential middleware mechanisms can be developed, composed, and for constrained examples verified tractably, using state of the art timed automata model checkers. However, to apply model checking to a wider range of real-time systems, particularly those involving more general forms of preemptive concurrency, new techniques are needed to address decidability and tractability concerns.

This paper makes three contributions to research on formal verification and validation of DRE systems. First, it describes how bounded fair scheduling policies introduce a quasi-cyclic structure in the state space of multi-threaded real-time systems. Second, it shows that bounds on the divergence of threads' execution can be determined for that quasi-cyclic structure, which then can be exploited to reduce the complexity of model checking. Third, it presents a case study involving progress-based fair scheduling of multi-threaded processing pipelines, with which the approach is evaluated.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Model Checking

General Terms: Verification, Scheduling

Keywords: Middleware, Timed Automata.

1. INTRODUCTION

Long running reactive service applications are a fundamental part of today's computing environments. These applications are often quasi-cyclic [9] in nature, endlessly repeating similar sets of activities according to temporal or system events. Usually, these applications (1) share common computing resources and middleware infrastructure to reduce system costs, (2) use multi-threading to support concurrent execution of activities, and (3) schedule threads preemptively to make such concurrency temporally predictable.

*Research at Washington University supported in part by NSF awards CCF-0615341 (EHS) and CCF-0448562 (CAREER).

EMSOFT'07, October 1–3, 2007, Salzburg, Austria.

Two major challenges in the verification of such systems are to represent their time and event semantics with high fidelity, and to apply analysis techniques that remain decidable and tractable across the wide variety of system concurrency and scheduling semantics that may arise in practice. Our previous research [17] has shown that incorporating domain-specific information into reusable formal models based on timed automata both increases the cost of analysis and offers new ways to mitigate that cost. For example, thread identifiers added to model run-to-completion semantics must be managed carefully to avoid over-constraining possible executions, but also allow the state space to be pruned accurately, e.g., for leader election in a thread pool [17].

In addition to the time and event semantics of the application and its supporting middleware, the effects of schedulers and other mechanisms for policy enforcement also must be considered in the verification of correct system behavior. Our previous work on verification of real-time systems considered a restricted but widely used class of scheduling policies (rate-monotonic scheduling of fixed-period tasks [17]) in which preemption only occurs at well defined points. However, many quasi-cyclic real-time systems have less constrained scheduling semantics. For example, in the example application we consider in more detail in Section 3, the relative progress of a related set of video processing pipelines may be kept within a specified bound [2], but may allow preemption points to vary dynamically as a function of application inputs or the system's operating environment.

The challenge posed by these more general scheduling semantics is that verifying correct behavior of systems with more widely dispersed preemption raises crucial questions, about the fidelity with which concurrency can be represented, and the decidability and tractability with which it can be analyzed. Furthermore, as we discuss in Section 2, there are important limitations on the ability of existing techniques to address these challenges, which in turn motivate the research presented in this paper.

In this paper we present a novel extension of quasi-cyclic state space reduction techniques [9] to timed systems, with the effect of allowing systems scheduled with bounded fairness to be verified tractably and with reasonable fidelity. In this approach, models of the individual process are composed into a single time domain with the result being an infinite state timed automaton (called a *time domain automaton*) in which each state represents an equivalence class of all possible execution interleavings that result in a particular event ordering. We then show that schedulers which enforce bounded fairness also prune the state space that is explored, giving a timed quasi-cyclic structure with well defined bounds.

The rest of this paper is structured as follows. Section 2 describes other work related to the approach presented in this paper. Section 3 discusses a video processing application that is representative

of the broader class of applications whose scheduling semantics motivate and guide our approach, and discusses the analysis challenges posed by that class of applications. Section 4 describes how timed automata models can be parameterized with domain-specific scheduling information, and how analytic bounds on the behavior of the system can be transformed into bounds on the structure of the state space that must be verified. Section 5 presents a case study in which we apply our parameterized modeling techniques to analyze the time and event semantics of the example application presented in Section 3, and evaluate how well our approach captures the actual behavior of that application. Finally, Section 6 presents conclusions and describes future work.

2. RELATED WORK

The most relevant general purpose modeling techniques (stopwatch automata [5], timed automata [1], and untimed finite automata) have important limitations for modeling preemptively scheduled systems: (1) checking stopwatch automata models may be undecidable, (2) the time representation in traditional timed automata is incompatible with the operations needed to model preemption, and (3) the resulting state space for untimed finite automata is intractably large.

Stopwatch automata, where clocks' derivatives can be either zero or one, would seem to be a natural way to model preemptively scheduled systems. In preempted states, the clock's derivative would be set to zero, and in a running state, the clock's derivative would be set to one. However because a system can change state infinitely many times between any two time values it has been shown that many scheduling problems are undecidable using stopwatch automata [14]. More powerful tools like Hytech [12], also allow verification of stopwatch automata, but in general may run into similar concerns with decidability and tractability.

Another approach for modeling systems with preemption is to use timed automata, a restricted form of stopwatch automata, where all clocks' derivatives must be one. Because clock values can take on any nonnegative real value, these systems have potentially infinitely many states. To allow state exploration to terminate, states must be collapsed into a finite set of equivalence classes. For timed automata model checkers like UPPAAL [3] and IF [4] this is done by using timed difference bound matrices [8] (TDBM).

To model preemptive scheduling the model checker must write out values of individual clocks and restore them later when a process is rescheduled. However, the TDBM stores disjunctions of inequalities that represent generally infinite clock values. Therefore, writing out the value of the clock requires picking a subset of possible clock values and splitting the current state into a number of states equal to the cardinality of the subset. The larger this subset the faster the propagation of states in the state transition system, leading quickly to intractability, while the smaller the subset, the less fidelity the model has to the modeled system.

A third approach for modeling systems with preemption is to discretize time. In this case, the system needs no clocks and can be modeled using finite automata. State exploration is now simply the task of assigning each quantum to a process. There are, however, exponentially many such orderings and thus full exploration takes exponential time: except for restricted systems or verification over small windows of execution, this method may be impractical.

The limitations of these general purpose modeling approaches have given rise to a number of other approaches, which capture and leverage additional information about the structure of the system itself. One such approach is to compose automata based on common interfaces and sets of resources [13, 6]. Another relevant approach is to identify quasi-cyclic structures in the system's execution, which can be used to reduce the state space that must be checked [9]. A third approach is to use abstract interpretation in combination with model checking [7] to reason about event interleavings and paths of execution in the system. While each of these approaches has influenced the work in this paper, none of them provides the combined ability to analyze relative resource consumption, timing, and preemption that is needed for verification of the kinds of systems we describe in Section 3.

3. MOTIVATING EXAMPLE

The research presented in this paper is motivated by the problem of verifying real-time systems in which (1) thorough analysis of the system's behavior with respect to timing and event ordering constraints is needed to verify its correctness; but (2) analysis is complicated by scheduling semantics that allow threads to be preempted at dynamically variable points in time, e.g., due to application-specific variations in task execution times, or environment-specific variations in wireless network bandwidth.

Figure 1 illustrates the high level architecture of such a system, which abstracts the structure of several specific applications that were the focus of our previous work [18, 2, 11]. The important

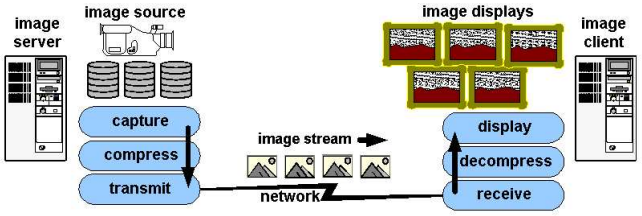

Figure 1: System Architecture

features of the system architecture include:

- server-side devices and/or repositories that capture and/or store streams of related images (e.g., frames of a video stream, or tiles of a large static image);
- multiple server-side tasks that capture, compress, and transmit each image;
- a network across which image streams are transmitted;
- multiple client-side tasks that receive, decompress, and display each image; and
- client-side devices where images are displayed.

Within the overall system architecture, alternative concurrency architectures and scheduling policies may be appropriate for different applications. For example if the image processing is a best effort activity that shares resources with other more critical processing [11], then a concurrency architecture that has different thread lanes for different levels of criticality, atop a priority-based thread scheduling mechanism [10] may be appropriate. This approach is well suited for enforcing a variety of classical real-time scheduling policies [10], including the rate-monotonic policy to which our previous verification approach was applied [17].

However, for other applications, such as a multi-camera video monitoring system for security, fire detection, and process control in an automated industrial plant, hardware may be dedicated explicitly to the capture, distribution, and processing of video streams. In this case, the challenge is less to isolate critical processing from interference by non-critical tasks, but rather to ensure synchronization and then reduce end-to-end latency of multiple related video

streams. For such an application, a pipelined concurrency model, in which multiple copies of each task, each with its own thread of execution (e.g., according to the Active Object pattern [15]) may be appropriate: (1) if a task is blocked on input or output, other tasks can be using the CPU and other resources to make progress; (2) tasks can be distributed or mapped to multiple processors easily if those additional resources are made available; and (3) all threads (and thus the execution of all tasks) can be placed under common scheduling control at either the operating system or middleware level [2].

Figure 2 illustrates such a concurrency architecture, with a separate processing pipeline connecting each server-side image source to a distinct client-side display. Each active object implements a

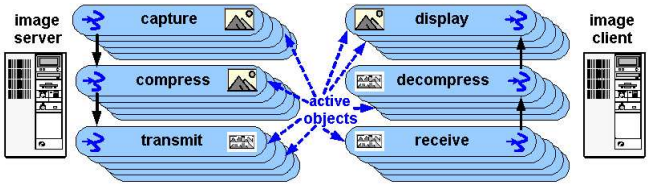

Figure 2: Concurrency Architecture

particular stage in a concurrent processing pipeline:

- multiple server-side video devices (or image repository searches) capture images concurrently;
- each captured image is then compressed concurrently with other images;
- compressed images are transmitted concurrently across the network;
- compressed images are received concurrently on the client;
- received images are decompressed concurrently;
- the client-side display for each pipeline is updated with its decompressed images, concurrently with the other pipelines.

While our previous work has shown that such a system can be built using open-source real-time operating systems and middleware, and that appropriate scheduling semantics can be enforced effectively and efficiently [2], verification of such a system remains an open problem despite our previous progress on modeling the semantics of the middleware mechanisms involved [17]. The remaining challenge is to develop a combined framework for evaluating the effects of different scheduling policies (the scheduling model) on different concurrency architectures (the process model) in such a system, in a manner that can support decidable and tractable reasoning about preemption and its related complications. Section 4 describes our development of such a framework, and Section 5 presents a verification case study using that framework, both of which are major contributions of this paper.

4. QUASI-CYCLIC TIMED STRUCTURES

To improve scalability of model checking for real-world systems, Dwyer, et al., introduced the idea of a *quasi-cyclic* structure, in which a predicate over system states produces a projection of the state space in which a set of sub-states – with the same values for a subset of the system's state variables – is visited recurrently [9]. Such quasi-cyclic structures occur naturally in many systems with cyclic timing and/or event semantics. When they do, their regular structure can make verification of these systems tractable by reducing the time and/or space needed to search the state space. However, each such quasi-cyclic projection is domain-specific, and must be constructed based on (1) a model of the system's states and (2) a bounding predicate whose constraint results in a quasi-cyclic projection. In this section we show how timed automata and fair scheduling can be combined to give quasi-cyclic structures for verification of preemptively scheduled systems.

Several extensions to the work in [9] are needed in order to take advantage of quasi-cyclic structures in preemptively scheduled systems. First, for real-time systems timed automata (rather than untimed automata) often are the most natural way to model system states. We use timed automata models to capture important features of preemptively scheduled processes, as is described in Section 4.1.

Second, the effects of preemption must be modeled in a way that is amenable to extracting a quasi-cyclic structure, i.e., allowing identification of common values of key variables (particularly regarding time). We do this by composing the individual process models into a single automaton with a common time domain, as is described in Section 4.2. This automaton in turn allows us to evaluate the timing and event ordering semantics resulting from the composition of the underlying process models, including the effects of preemption on the possible event orderings in the system.

Third, the quasi-cyclic structure must be extracted based on constraints on system execution that bound the possible event structures. In Section 4.3 we focus specifically on how different constraints produce quasi-cyclic projections that can reduce the complexity of verification for systems like those in Section 3.

For clarity, throughout this section we illustrate key features of our approach using only a pair of processes. However, the approach and equations presented here generalize to arbitrary numbers of processes. We also evaluate the approach with a case study involving five processes, in Section 5.

4.1 Process Models

In this paper we use the term *process* to refer to any modular unit of execution, such as a single thread or a group of related threads (e.g, within a pipeline as shown in Figure 2 in Section 3). We start by capturing key features of system processes as timed automata, as is shown for two process in Figure 3. The transitions represent

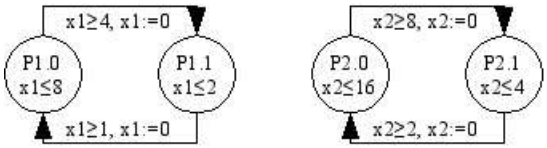

Figure 3: Process Automata P1 and P2

distinct events in the system, the process models' states (P1.0, P1.1, P2.0, and P2.1) record execution times (x1 and x2), and the guards on each transition and the invariants in each state represent minimum and maximum bounds on the demand function [16] of each process.

4.2 Time Domain Automaton

We use *time domains* to represent multiple automata sharing a single resource and the resulting possible event structures and their time boundaries. To compose the automata shown in Section 4.1 into the same time domain, τ, we now make use of their guards and invariants. The composition of these two automata into the same time domain generates a single infinite state timed automaton, which for sake of discussion we call a *time domain automaton*. This automaton represents every feasible ordering of events and

the invariants and guards that govern when, given the underlying process structure, these events could occur. Figure 4 shows the first states of the time domain automaton resulting from composing the process automata shown in Figure 3.

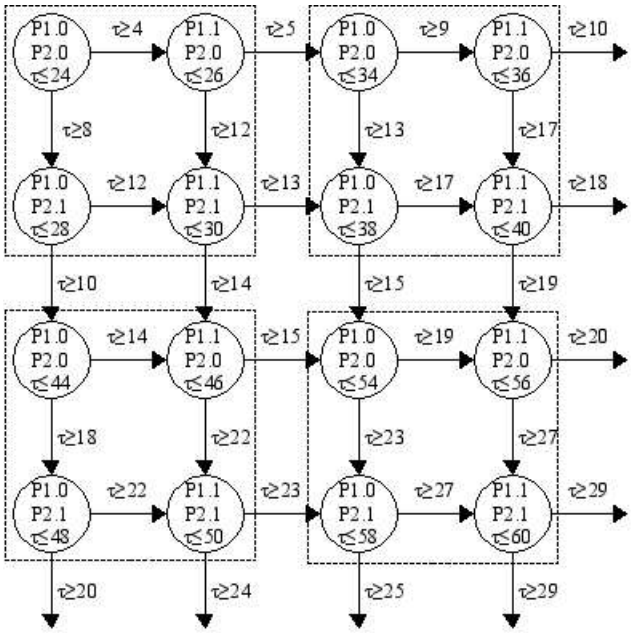

Figure 4: P1||P2 in a Single Time Domain τ

Each edge in the time domain automaton is an edge from one of the composed process automata. Each state in a time domain automaton is described by the set of states each process automaton would be in and the possible values of the following variables: τ (the system time) and p_i (the relative time for process i). Because τ is the sum of the values for the individual processes, the bounds for τ can be calculated from the bounds on each p_i as follows:

$$A_1 \leq p_1 \leq B_1$$

$$\vdots$$

$$A_k \leq p_k \leq B_k$$

$$\sum_{i=1}^{k} A_i \leq \tau \leq \sum_{i=1}^{k} B_i$$

The bounds on each p_i are functions of the path from the initial state to the current state in the time domain automaton. This path can be projected onto the individual process models.

Intuitively, the minimum bound for p_i is the minimum possible computational resource needed for the process to make the progress it has made. Mathematically, it is a function of the path taken in the time domain automaton to the current state. A subset of the edges in this path will correspond to an edge in process automaton i, although the subset might be empty. Each edge in this subset is associated with a guard in the process automaton. The minimum bound is the sum of these associated guard values for each edge in the subset.

Similarly, the maximum bound for p_i is the maximum computational resource that could be granted to the process without guaranteeing it will move to a new state. It too is a function of the path taken to the current state. Mathematically, it is the sum of the invariants for each state in the process model that the time domain automaton has passed through, including the current state.

Despite having infinitely many states, this time domain automaton can be treated as a normal timed automaton for the purposes of composition with other timed automata. In addition, composition of separate time domain automata is trivial, because as can be seen in Figure 4, the resulting automaton has no clock resets: after composition time is still represented by a single variable.

Without a way of enforcing an equivalence partition over the states in the time domain automaton, full exploration is impossible. However, full exploration to any finite τ is possible, but even for these more limited searches, the state space grows exponentially as time moves away from the origin. This is partially offset by the fact that the time domain automaton is quasi-cyclic, as illustrated by the rectangles around related states in Figure 4. Following [9], the quasi-cyclic nature of time domain automata can be exploited to reduce the space requirements dramatically for a state space exploration out to any given value of τ.

4.3 Constraints and Quasi-Cyclic Structures

Time domain automata capture the semantics of all possible system event orderings, while constraints model system behaviors that bound possible event orderings. As examples, we discuss three forms of constraints: (1) fairness constraints, (2) bounded delay constraints and (3) event based constraints.

Fairness constraints: In each state bounds exist for each process' execution time as well as the total system execution time. The ratio of process execution time to total system time represents the *fairness* of the state, and its bounds can be calculated for a state as a whole. To handle cases where the time domain automaton represents a resource partition of less than 100%, we introduce σ_{min} and σ_{max} which represent the minimum and maximum resource allocations, which in turn constrict the fairness values to the range [0,σ_{max}]. Equation 1 shows the fairness bounds for a state as a whole.

$$\frac{\sigma_{min} A_i}{A_i - B_i + \sum_{j=1}^{k} B_j} \leq \frac{p_i}{\tau} \leq \frac{\sigma_{max} B_i}{B_i - A_i + \sum_{j=1}^{k} A_j} \qquad (1)$$

The minimum and maximum fairness values are located at one of the boundary points of the variable space that describes the state. Because the variables take on only positive, real values, minimizing the lower fairness bound simply requires minimizing all values that appear only in the numerator while maximizing those that appear only in the denominator. For most real systems a set of fairness constraints must be satisfied, often taking the form of a conjunction of inequalities. As generated, each state in the time domain automaton can be tested, and exploration can be stopped at any state whose fairness values contradict the fairness constraints. The remaining subset, while still potentially infinite, captures all event structures that could achieve the fairness constraints.

Bounded delay constraints: Another kind of constraint commonly imposed on real systems bounds the delay between two events. Consider the constraint 'P1.1 transitions to P1.0 followed by P2.1 transitioning to P2.0 within 40 time units.' To explore what event structures are possible under this constraint, and thus to gauge its effect on the full time domain automaton, we construct a new automaton, regions of which are untimed, and others which are generated like a time domain automata.

Until the system sees the first event covered by the constraint, the possible timings of the events in the system are inconsequen-

tial. Once the event occurs, we generate states as we would for a time domain automaton until one of two conditions occurs: (1) the possible values of τ in the state and the constraint condition are in contradiction (e.g. $\tau \geq 44$ and $\tau \leq 40$) or (2) the second event covered by the constraint occurs and we return to the untimed region of the automaton. Each bounded delay constraint generates an automaton, and these automata can be composed to create a single automaton that represents the conjunction of all constraints.

Event based constraints: Another type of constraint can be expressed as an untimed finite state automaton, which captures allowed event interleavings directly. Examples of this type of constraint and their effects are described next in Section 4.4. No matter what kind of constraint or set of constraints is applied, the result is a state space that is a subset of the full time domain automaton. These constraints result in a bounded time domain automaton with an induced quasi-cyclic state space. Model checking of the bounded time domain automaton can be done to any finite value of τ to ensure that (1) its state space is non-empty, (2) it has satisfying states that extend out to the given τ value, and (3) desired properties within the state-space are maintained.

4.4 Fair Scheduling Example

To illustrate how the modeling approach described in Sections 4.1 through 4.3 applies to a real-world system, we now give a basic example based on the system architecture and concurrency architecture of the video processing pipeline system illustrated in Figures 1 and 2 in Section 3. Although for purposes of illustration we concentrate on the case of only two pipelines, the examples presented here are the basis for the five pipeline example used in the evaluation in Section 5.

We begin with a process model for each pipeline as shown in Figure 5. Each process i is a simple one state automaton whose upper and lower execution times are bounded by α_i and β_i, respectively.

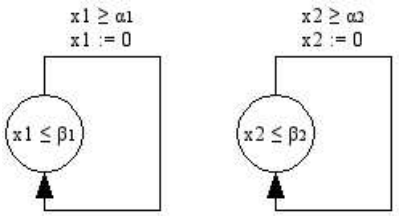

Figure 5: Video Pipeline Process Model

We then model a scheduling decision function (SDF) that ensures that each video pipeline publishes frames in lockstep, such that their relative progress is kept within a specified bound of at most one frame (where one frame is processed by each execution of an entire pipeline). As in our previous work we will consider the case where this bound is a single frame. A simple SDF can be designed to enforce this behavior directly, by defining a constraining automaton (which we call the *SDF automaton*) over the frame processing events. The SDF automaton for constraining the two pipelines case is shown on the left side of Figure 6, and for purposes of illustration the right side of Figure 6 also shows the SDF automaton for the three pipelines case.

Each state in an SDF automaton is labeled with an array a (of length equal to the number of pipelines) that shows the relative frame progress of each pipeline. As pipeline i completes a frame the SDF transitions to the state labeled with a 1 in the ith slot in the array, or back to the origin if all other pipelines have completed their frame for the current iteration. For the general case

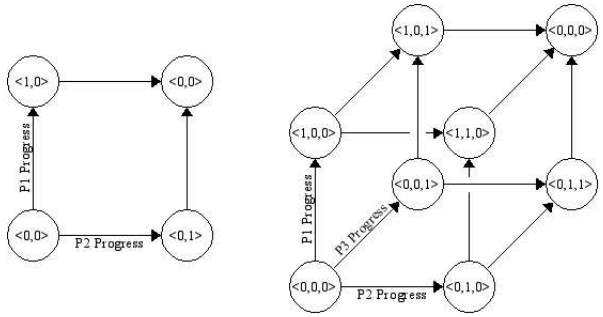

Figure 6: 2-Process and 3-Process SDFs

of k pipelines, the SDF automaton is a k-dimensional hyper-cube with $2^k - 1$ distinct states (because the first and final states are identical).

Using time domain automata we can examine the effects of applying this SDF to the running pipelines. For simplicity we first look at the effect of the SDF on the 2 pipeline example, and then extend it to arbitrary numbers of pipelines.

As we have seen before, the state space of the resulting time domain automaton is quasi-cyclic. If we project a path in that time domain automaton onto the 2 process SDF automaton in Figure 6, the path cycles through the SDF automaton some number of times. This value, which we call n, can be used to partition the states in the time domain automaton. As is shown in Figure 7 the bounds for p_i in each state in a partition is a linear function of n. Because we can write the bounds for p_i as linear functions of n the same holds true for the fairness bounds for process i. These equations are shown in Figure 8, including their convergence as n approaches infinity.

From the 2 pipeline example we can derive general equations for k pipelines. As we saw earlier, each state is labeled with an array, and the time domain automaton that results from applying the SDF can be partitioned with the variable n. That means every state in the time domain automaton can be uniquely identified by the value of n and the array representing the current state in the SDF automaton. General equations for p_i can be written in terms of n and $a[i]$, the ith entry in the array labeling the state in the SDF automaton. This is shown in Equation 2.

$$(n + a[i])\alpha_i \leq p_i \leq (n + a[i])\beta_i + \beta_i \tag{2}$$

We can also derive general fairness equations for the k pipeline case by using the derived p_i bounds in conjunction with the fairness equation introduced previously (Equation 1). From these we can derive general formulas for the fairness bounds as n approaches infinity, shown as Equation 3.

$$\frac{\sigma_{min}\alpha_i}{\alpha_i - \beta_i + \sum_{j=1}^{k} \beta_j} \leq \frac{p_i}{\tau} \leq \frac{\sigma_{max}\beta_i}{\beta_i - \alpha_i + \sum_{j=1}^{k} \alpha_j} \tag{3}$$

Another interesting feature of this constraint automaton is that in some states one or both the fairness bound equations are independent of n and equal to the limit case fairness bound equation. These states are described in Equations 4 and 5. The states described by Equation 4 are states in which process i has just finished publishing the first frame in the current iteration. The states described by Equation 5 are states in which process i is the last pipeline not to

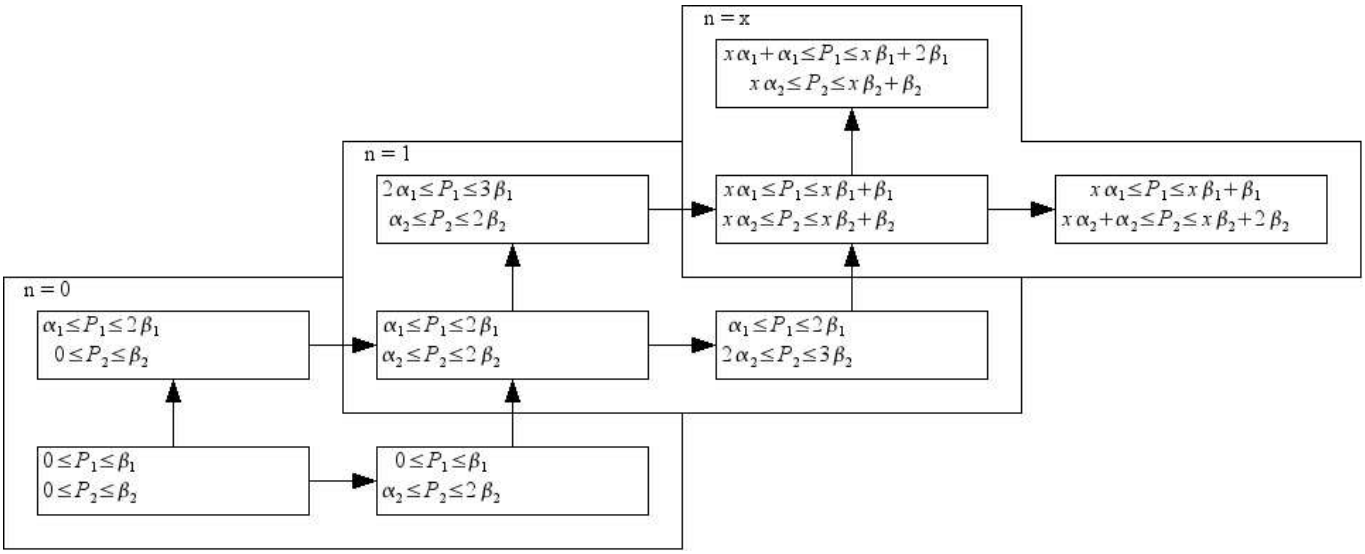

Figure 7: Relative Time Bounds for 2-Process Composition

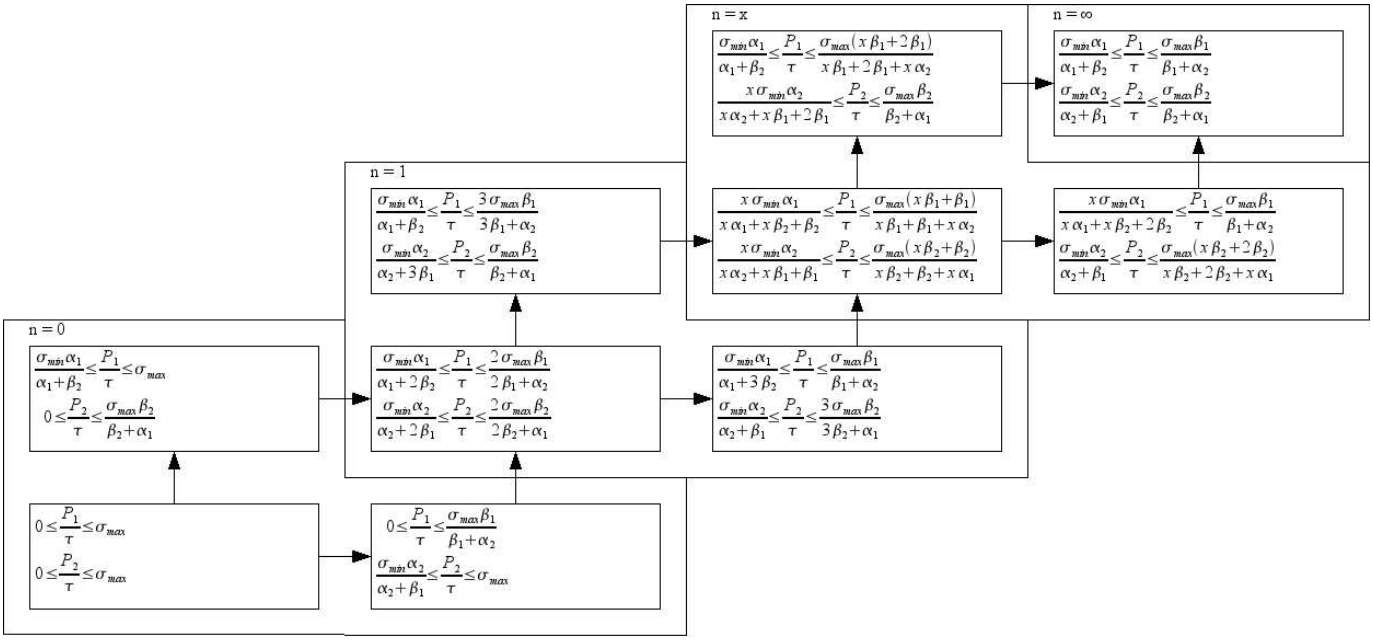

Figure 8: Fairness Bounds for 2-Process Composition

have published its frame in the current iteration.

$$\frac{\sigma_{min}\alpha_i}{\alpha_i - \beta_i + \sum_{j=1}^{k} \beta_j} \leq \frac{p_i}{\tau} \text{ if } a[j \neq i] = 0 \text{ and } a[i] = 1 \quad (4)$$

$$\frac{p_i}{\tau} \leq \frac{\sigma_{max}\beta_i}{\beta_i - \alpha_i + \sum_{j=1}^{k} \alpha_j} \text{ if } a[j \neq i] = 1 \text{ and } a[i] = 0 \quad (5)$$

5. CASE STUDY

To evaluate the applicability of the modeling framework presented in Section 4 to the kinds of systems described in Section 3, we now consider a case study consisting of five video pipelines running on a single CPU, scheduled according to the frame-progress fair scheduling policy discussed in Section 4.4. Because the resulting quasi-cyclic structure has 31 distinct states we do not depict it, but simply note that it is a 5-dimensional generalization of the cube shown for three processes in Figure 6.

To illustrate the independence of the individual processes in the quasi-cyclic structure, and to demonstrate the specificity of the scheduling decision function, we assigned some common and some distinct demand bounds to the processes: pipelines 1 and 3 used between 70 and 80 msec of execution each time one of them ran; pipelines 2 and 4 used between 80 and 90 msec of execution each time one of them ran; pipeline 5 used between 100 and 110 msec of execution each time it ran. The processing of each pipeline was triggered every 500 msec, so the total utilization is thus between 80% and 90%, i.e., reasonably heavily loaded but feasibly schedulable. Finally, the actual demand for each execution of a pipeline was a normally distributed random sample in the range from that pipeline's lower bound to its upper bound.

We used the processes' execution time bounds directly as their α and β parameters, as is summarized in Table 1. These parameters are used to compute minimal and maximal progress bounds for each process over the state space, as follows. The first state in the quasi-cyclic structure is labeled with array a = <0,0,0,0,0>. The scheduler-enforced utilization is between 80% and 90%, so the fairness bounds are calculated using $.80 = \sigma_{min}$ and $.90 = \sigma_{max}$. Using Equation 2, we calculate bounds for each p_i.

Table 1: Pipeline Progress Bounds

Pipeline	α	β
1	70	80
2	80	90
3	70	80
4	80	90
5	100	110

$$70n \leq p_1 \leq 80n + 80$$

$$80n \leq p_2 \leq 90n + 90$$

$$70n \leq p_3 \leq 80n + 80$$

$$80n \leq p_4 \leq 90n + 90$$

$$100n \leq p_5 \leq 110n + 110$$

With these bounds, σ_{min}, σ_{max}, and Equation 3 we calculate the specified fairness bounds for the system.

$$\frac{.80(70n)}{440n + 370} \leq \frac{p_1}{\tau} \leq \frac{.90(80n + 80)}{410n + 80}$$

$$\frac{.80(80n)}{440n + 360} \leq \frac{p_2}{\tau} \leq \frac{.90(90n + 90)}{410n + 90}$$

$$\frac{.80(70n)}{440n + 370} \leq \frac{p_3}{\tau} \leq \frac{.90(80n + 80)}{410n + 80}$$

$$\frac{.80(80n)}{440n + 360} \leq \frac{p_4}{\tau} \leq \frac{.90(90n + 90)}{410n + 90}$$

$$\frac{.80(100n)}{440n + 340} \leq \frac{p_5}{\tau} \leq \frac{.90(110n + 100)}{410n + 110}$$

In practice, tighter bounds on the system execution may be available empirically. For instance, the observed utilization by the pipelines in our experiments was 87.35%. Setting both σ_{min} and σ_{max} to that value, we can derive empirical bounds, which are tighter than the specified bounds. Note that the specified bounds are still necessary for verification, because the empirical bounds may not reflect the system's worst case behavior.

We compared the derived bounds to the actual behavior of a 5-pipeline system under middleware-based fair-progress group scheduling control. For those experiments, we used the test application and middleware-based group scheduling framework described in [2], configured with a frame-progress-fair scheduling decision function, run atop a Linux 2.6.12 kernel with the KURT-Linux and RT (version 0.7.50-04) patches. Figure 9 shows both sets of bounds and the measured progress of each process. Other than the points at which the scheduler ensures that all pipelines have made consistent progress, any particular pipeline may wait until the very last state of the quasi-cyclic structure to make any progress, or may make its entire progress in the first state of the quasi-cyclic structure. Thus, the progress bounds are step functions with the maximal bound increasing only in the first state of the quasi-cyclic structure, and the minimal bound increasing only in the last state of the quasi-cyclic structure. The observed behavior of the processes falls within these analytical bounds, as expected.

Because the semantics of the particular scheduling decision function used in these experiments is to ensure that each pipeline completes processing of an image frame before any are allowed to begin processing their next image frame, it is also useful to examine the behavior of the system relative to the calculated bounds at those enforced scheduling points. Figure 10 shows an abstraction of the data presented in Figure 9 consisting only of the enforced scheduling points. As Figure 9 shows, the progress within a given quasi-cyclic structure need not be significantly constrained, other than by coarse overall bounds, and thus model checking of the behavior within each stage of a quasi-cyclic structure may be appropriate for some systems. However, for systems that are more tolerant of minor variations in progress, the abstraction illustrated in Figure 10 may suffice, in which case the cost of verification is reduced to an analysis of the bounds enforced by the scheduler (and possibly a verification of the scheduler itself).

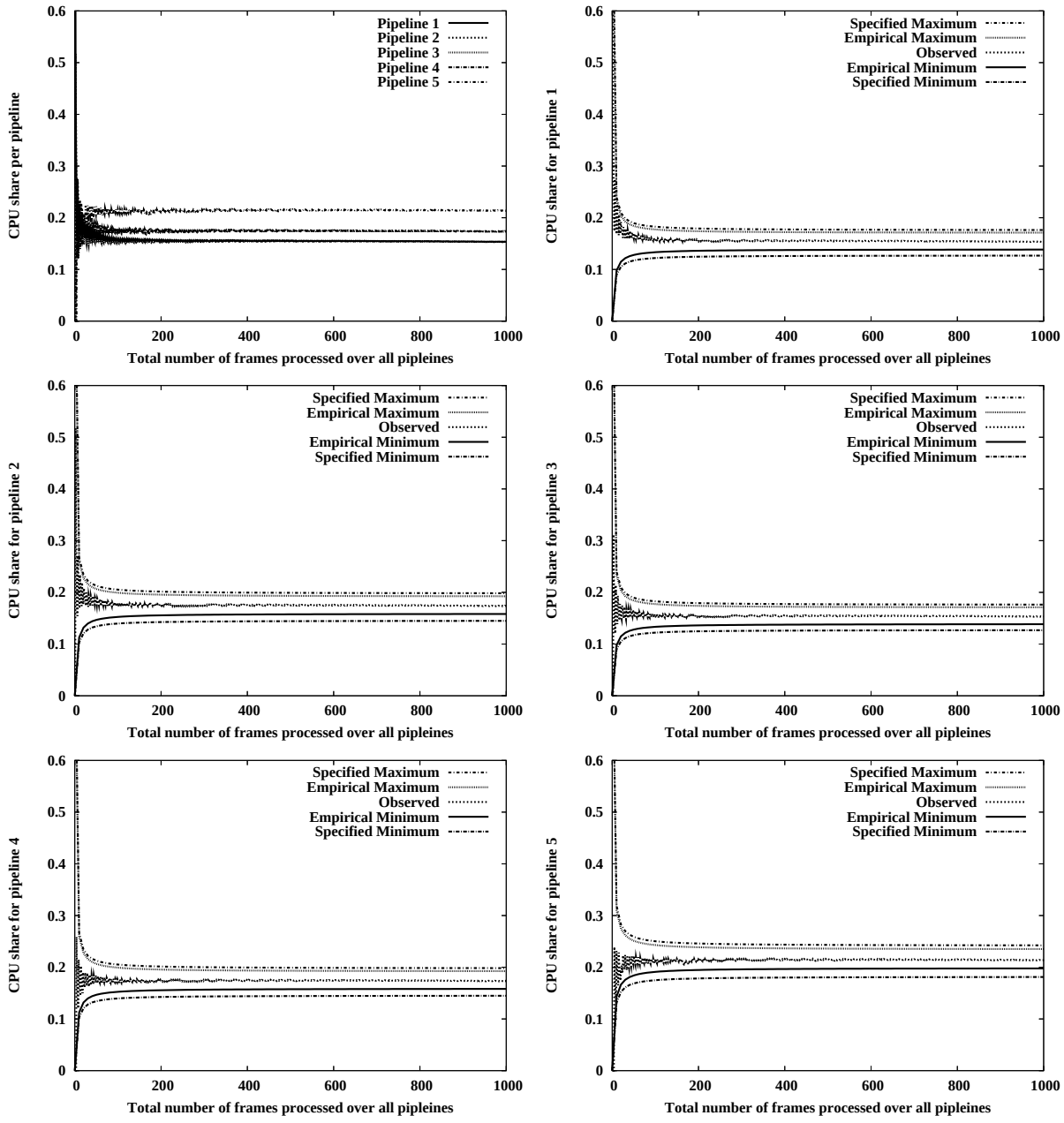

Figure 9: Total Progress and Calculated Bounds for all Processing Pipelines

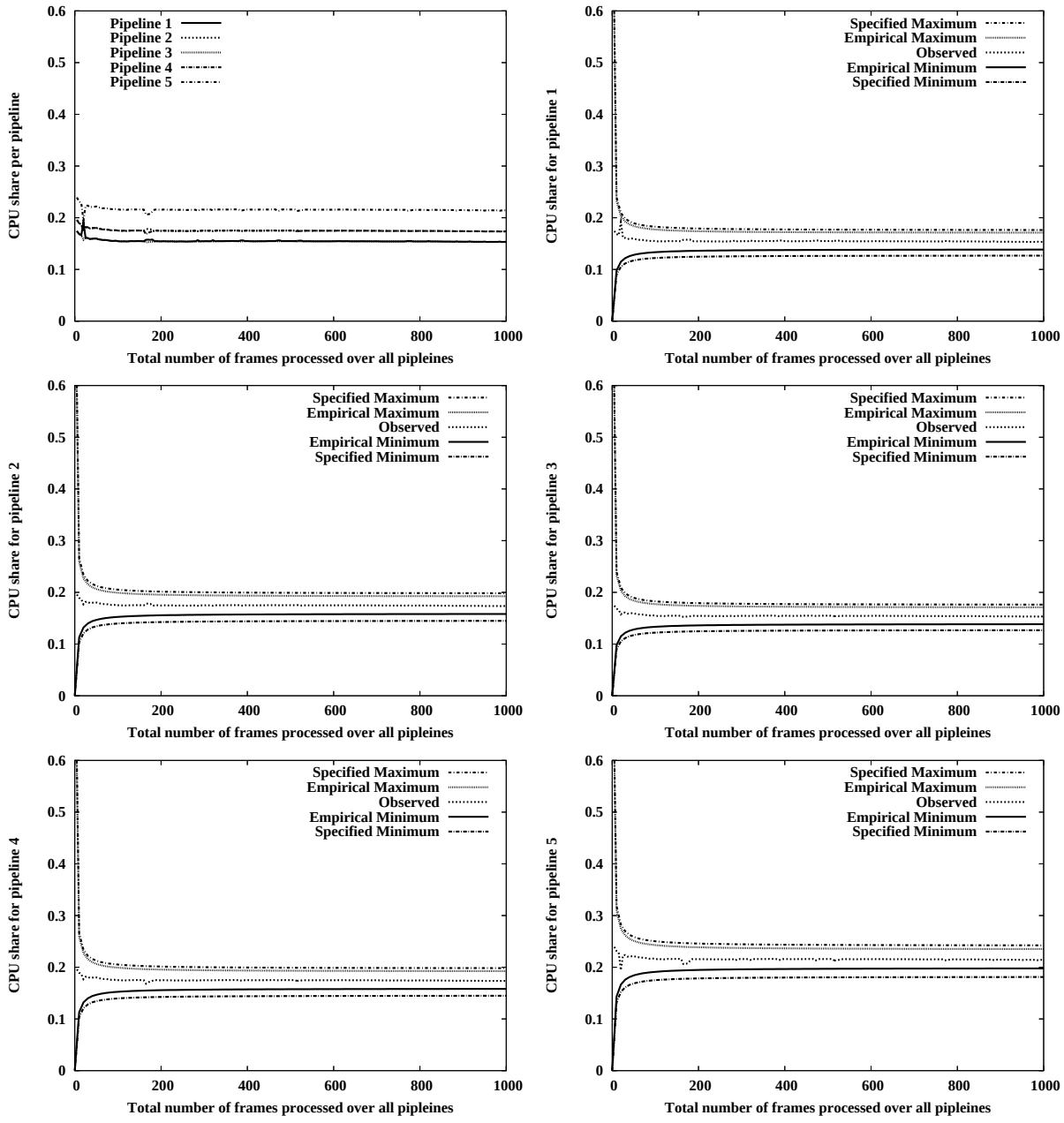

Figure 10: Observed Progress and Calculated Bounds at Scheduling Points

6. CONCLUSIONS AND FUTURE WORK

In this paper we have discussed several ways in which quasi-cyclic structures can be identified and exploited in preemptively scheduled real-time systems. Our modeling approach presented in Section 4 supports analysis based on a variety of different kinds of constraints, including fairness constraints, bounded delay constraints, and event based constraints. Our case study in Section 5 showed how our approach can be applied to verification of a particular kind of system with a unique concurrency architecture and scheduling semantics.

Our future work will focus on the nuances of state space exploration under different kinds of domain-specific constraints. In particular, we will investigate (1) the semantics of different combinations of fairness constraints, bounded delay constraints, and timing constraints; and (2) how quasi-cyclic structures can be identified and leveraged in verification of systems with different combinations of those constraints.

Acknowledgments

We wish to thank Dr. Douglas Niehaus for developing the Group Scheduling model, through which the progress-fair scheduling semantics were enforced in our prior work [2], and in the evaluations we presented in Section 5.

7. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] T. Aswathanarayana, V. Subramonian, D. Niehaus, and C. Gill. Design and performance of configurable endsystem scheduling mechanisms. In *Proceedings of 11th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[3] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-time Systems (SFM 2004)*, pages 200–236. Springer-Verlag LNCS 3185, Sept. 2004.

[4] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. In *Formal Methods for the Design of Real-time Systems (SFM 2004)*, pages 237–267. Springer-Verlag LNCS 3185, Sept. 2004.

[5] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *11th International Conference on Concurrency Theory (CONCUR 2000)*, pages 138–152, Aug. 2000.

[6] A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *Third ACM International Conference on Embedded Software (EMSOFT 2003)*, pages 117–133, Oct. 2003.

[7] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. Technical Report SAnToS-TR2006-1, Laboratory for Specification, Analysis, and Transformation of Software (SAnToS), Kansas State University, 2006.

[8] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212. Springer-Verlag LNCS 407, 1990.

[9] M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Third ACM International Conference on Embedded Software (EMSOFT 2003)*, pages 173–189, Oct. 2003.

[10] C. Gill, D. C. Schmidt, and R. Cytron. Multi-Paradigm Scheduling for Distributed Real-time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), Jan. 2003.

[11] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt. Integrated Adaptive QoS Management in Middleware: An Empirical Case Study. *Journal of Real-time Systems*, 29(2–3):101–130, 2005.

[12] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In *Hybrid Systems: Computation and Control (HSCC 2000)*, pages 130–144, Mar. 2000.

[13] T. A. Henzinger and S. Matic. An interface algebra for real-time components. In *Proceedings of 12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2006.

[14] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Decidable integration graphs. *Information and Computation*, 150(2):209–243, 1999.

[15] R. G. Lavender and D. C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, pages 1–7, Monticello, Illinois, Sept. 1995.

[16] I. Shin and I. Lee. Compositional Real-Time Scheduling Framework. In *The 25th IEEE Real-time Systems Symposium (RTSS)*, Lisbon, Portugal, Dec. 2004.

[17] V. Subramonian, C. Gill, C. Sánchez, and H. B. Sipma. Reusable models for timing and liveness analysis of middleware for distributed real-time and embedded systems. In *Sixth ACM/IEEE International Conference on Embedded Software (EMSOFT 2006)*, pages 252–261, Oct. 2006.

[18] X. Wang, H.-M. Huang, V. Subramonian, C. Lu, and C. Gill. CAMRIT: Control-based Adaptive Middleware for Real-time Image Transmission. In *Proc. of the 10th IEEE Real-time and Embedded Tech. and Applications Symp. (RTAS)*, Toronto, Canada, May 2004.