Report Number: WUCSE-2008-1

2008-01-01

Modeling Timed Component-Based Real-time Systems

Huang-Ming Huang and Christopher Gill

Component based middleware helps to facilitate software reuse by separating application-specific concerns into modular components that are shielded from the concerns of other components and from the common concerns addressed by underlying middleware services. In real-time systems, concerns such as invocation rates, execution latencies, deadlines, and concurrency semantics cross-cut multiple component and middleware abstractions. Thus, the verification of these systems must consider features of the application components (e.g., their execution latencies and relative invocation rates) and of the supporting middleware (e.g., concurrency and scheduling) together. However, existing approaches only address a sub-set of the features that must be modeled... Read complete abstract on page 2.

Modeling Timed Component-Based Real-time Systems

Huang-Ming Huang and Christopher Gill

Complete Abstract:

Component based middleware helps to facilitate software reuse by separating application-specific concerns into modular components that are shielded from the concerns of other components and from the common concerns addressed by underlying middleware services. In real-time systems, concerns such as invocation rates, execution latencies, deadlines, and concurrency semantics cross-cut multiple component and middleware abstractions. Thus, the verification of these systems must consider features of the application components (e.g., their execution latencies and relative invocation rates) and of the supporting middleware (e.g., concurrency and scheduling) together. However, existing approaches only address a sub-set of the features that must be modeled in component based real-time systems, and a new more comprehensive approach is needed. To address that need, this paper offers three main contributions to the state of the art in the verification of component based real-time systems: (1) it introduces a formal model called component automata that combines new input/output rate specifications with input/output actions and timed internal actions from the existing interface automata and timed automata models respectively; (2) it presents new component composition operations for single-threaded and cooperative multi-tasking, in addition to composition under the preemptive multi-tasking semantics assumed by interface automata; and (3) it describes how the composed component models then can be combined with task location specifications, a scheduling model, and a communication delay model, to generate a combined timed automaton representation of the components and middleware that can be verified by existing timed model checkers. This research was supported in part by NSF grant CCF-0448562 titled CAREER: Time and Event Based System Software Construction.

2008-1

Modeling Timed Component-Based Real-time Systems

Authors: Huang-Ming Huang and Christopher Gill

Corresponding Author: cdgill@cse.wustl.edu

Web Page: http://www.cse.wustl.edu/~cdgill/CAREER/

Abstract: Component based middleware helps to facilitate software reuse by separating application-specific concerns into modular components that are shielded from the concerns of other components and from the common concerns addressed by underlying middleware services. In real-time systems, concerns such as invocation rates, execution latencies, deadlines, and concurrency semantics cross-cut multiple component and middleware abstractions. Thus, the verification of these systems must consider features of the application components (e.g., their execution latencies and relative invocation rates) and of the supporting middleware (e.g., concurrency and scheduling) together. However, existing approaches only address a sub-set of the features that must be modeled in component based real-time systems, and a new more comprehensive approach is needed.

To address that need, this paper offers three main contributions to the state of the art in the verification of component based real-time systems:
(1) it introduces a formal model called component automata that combines new input/output rate specifications with input/output actions and timed internal actions from the existing interface automata and timed automata models respectively; (2) it presents new component composition operations for single-threaded and cooperative

Notes:
This research was supported in part by NSF grant CCF-0448562 titled CAREER: Time and Event Based System Software Construction.
Type of Report: Other

Modeling Timed Component-Based Real-time Systems*

Huang-Ming Huang and Christopher Gill
Department of Computer Science and Engineering
Washington University, St.Louis, MO, USA
{hh1, cdgill}@cse.wustl.edu

Abstract

Component based middleware helps to facilitate software reuse by separating application-specific concerns into modular components *that are shielded from the concerns of other components and from the common concerns addressed by underlying middleware services. In real-time systems, concerns such as invocation rates, execution latencies, deadlines, and concurrency semantics cross-cut multiple component and middleware abstractions. Thus, the verification of these systems must consider features of the application components (e.g., their execution latencies and relative invocation rates) and of the supporting middleware (e.g., concurrency and scheduling) together. However, existing approaches only address a sub-set of the features that must be modeled in component based real-time systems, and a new more comprehensive approach is needed.*

To address that need, this paper offers three main contributions to the state of the art in the verification of component based real-time systems: (1) it introduces a formal model called component automata *that combines new input/output rate specifications with input/output actions and timed internal actions from the existing interface automata and timed automata models respectively; (2) it presents new component composition operations for single-threaded and cooperative multi-tasking, in addition to composition under the preemptive multi-tasking semantics assumed by interface automata; and (3) it describes how the composed component models then can be combined with task location specifications, a scheduling model, and a communication delay model, to generate a combined timed automaton representation of the components and middleware that can be verified by existing timed model checkers.*

1. Introduction

To promote the separation of application-specific and common concerns in real-time systems, new forms of real-time middleware[19, 22] have emerged which typically offer flexible options for timers, threading, remote communication, and other common features, which can be configured specifically for each application's needs. Unfortunately, the very flexibility that allows desirable combinations of component and middleware features to be configured, also may allow configurations in which deadlocks, race conditions, missed deadlines, and other concurrency and timing hazards can arise. Furthermore, a middleware configuration that is suitable for one set of applications may introduce hazards for a different set of applications. Although the concerns encapsulated by individual components and middleware services are usually documented by their developers, concerns easily can be overlooked by system integrators during the component assembly process and as an application grows larger, the increasing number of components may cause an explosion of possible combinations of configuration options, making manual verification impractical.

Therefore, it is essential to develop automated tools for verification of these systems. The tools should track the compatibility of software components, provide valid middleware configuration options, and verify the presence, absence, or possibility of properties such as deadlocks or the timeliness of required responses. Model checking has emerged as an important technology for verification of real-time systems in which application and middleware details must be analyzed together, but no existing model checking approach is entirely well suited for verification of systems built with real-time component middleware. Section 2 summarizes work related to the research presented in this paper, and compares our work to those approaches.

Contributions of this paper: To address the limitations of existing approaches for verification of systems built using real-time component middleware, this paper offers a formal verification approach that is specifically designed for those systems. Section 3 provides an overview of the approach along with a brief discussion of the timed automata model upon which the approach builds. This paper provides three main contributions to the state of the art in verification of component-based real-time systems: (1) Section 4 introduces a formal model called *component automata* that combines new input and output rate relationships with in-

*This research was supported in part by NSF grant CCF-0448562 titled "CAREER: Time and Event Based System Software Construction."

put/output actions and timed internal actions (from the existing interface automata and timed automata models respectively); (2) Section 5 presents new component composition operations for single-threaded and cooperative multitasking, along with composition under the multi-threaded semantics assumed by interface automata; and (3) Section 6 describes how the composed component models then can be combined with task location specifications, a scheduling model, and a communication delay model, to generate a timed automaton representation of the combined components and middleware that can be verified by existing timed model checkers. Section 7 summarizes these contributions and offers concluding remarks.

2. Related Work

Component modeling environments: A body of ongoing research has focused on how to ensure the correctness of component based software systems. Karsai et al. [9] proposed a model-integrated approach for software development in which formal domain specific models are used within a software development process.

In Ptolemy [8] *actors* communicate through interfaces called *ports*. The execution of atomic actors is described in terms of interface automata [6]. The PTIDES [23] approach includes an executable simulation capability, but unlike our approach does not support an executable composition with models lower level middleware components.

DREAM [14] supports model-based schedulability analysis of time and event-driven DRE systems. DREAM offers a computational model consisting of tasks, timers, event channels and schedulers. Tasks are triggered either by a timer or external aperiodic events and tasks communicate among themselves by means of an event channel. Within this computational model, DREAM considers the problem of deciding the schedulability of a given set of tasks with time and event-driven interactions. By using timed automata models for each of the elements in the computational model, the schedulability problem is converted [14] into a reachability problem in the composed model.

Formal models: Model checking is a powerful approach for the automatic verification of finite state concurrent and reactive systems. Generally speaking, a system to be checked is modeled as a state transition system which can be converted to finite state automata (e.g. Büchi automata [21]). Traditional model checkers like SPIN [12] and Bogor [15] do not support explicit modeling of time. In other words, specifying the relative magnitude of delays between events, which may be critical to verifying correctness of real-time systems such as aircraft, industrial machinery and robots, is not directly supported in those tools. Several approaches have been proposed toward addressing model checking real-time systems, by modeling time explicitly. The first approach is discrete time modeling, in which a global non-decreasing clock is maintained and monotonically incremented [20] [5]. All automata in the system can read and compare local clocks against the global clock to calculate the relative delays between two states. The benefit of this approach is that it can be integrated easily with traditional model checking tools. BIP[2] is an example of a real-time component modeling framework built on top of the discrete time model. However, the discrete time model requires that continuous time be approximated by a fixed quantum (in advance) which may limit the precision with which the system is modeled.

The other approach for modeling time is the dense time model. In this model, times at which events occur are represented as real numbers which increase monotonically without bound. The representative formalization of this model is called *timed automata* [1] which we review in the next section. Although timed automata allow modeling of dense time, it is not possible to express preemption semantics in a timed automata model. More specifically, the flow conditions of the variables in a timed automata model must remain constant in all states. In other words, it cannot directly model and verify the behavior of a system with preemptive scheduling policies. Hybrid automata [10] constitute another formal model for mixed discrete-continuous systems where the flow conditions of variables can change among states. Therefore, it is possible to represent preemption behaviors by setting the flow conditions of certain variables in some states to zero. One drawback of hybrid automata is that their verification is generally undecidable except with some special constraints, and the complexities of those decidable special cases are often NP-hard.

Modeling middleware services: The mapping between software components and the automata for model checking is also an important topic. One way to model component based applications and their supporting middleware services is for each software component to be modeled as an individual automaton and the communications between components to be represented by channels in various representations supported by modeling checking tools; however, this approach does not fully capture the semantics of the application when components can be collocated on the same host. The problem arises in the context of the reactor or leader/follower patterns [17] that are used in the design of most middleware service layers (for the sake of memory and CPU efficiency). As described by Subramonian et al. [18], the use of those patterns coupled with different configuration options (such as wait-on-reactor or wait-on-connection) in middleware, can affect the safety and liveness properties of a system.

In [18], Subramonian et al. demonstrated techniques that support middleware modeling in UPPAAL and the IF toolset. These techniques map software abstractions directly to

timed automata. For example, inter-process communication (IPC) channels are modeled with a set of read/write buffers, and read/write operations of the IPC channel model are directly invoked. Although this approach epitomizes the actual implementation of software systems, it suffers from three problems: (1) lack of higher-level abstractions – model developers must specify the communication in terms of read/write operations on the IPC channels, which is contrary to the general principle of encapsulation; (2) it contains many details which may not be essential for modeling and model checking at the application level, and thus is more prone to inflict state space explosion [7]; and (3) every software component is treated as an active object [17] which creates the potential for mismatches between models and different concurrency implementations and makes models more difficult to develop and understand.

3. Overview of the Solution Approach

As was described in the previous section, there are important limitations of the existing modeling approaches. Timed automata do not support preemption, interface automata lack a way to specify relative rate relationships between input and output, and model checking with hybrid automata is generally undecidable. Our approach combines and extends timed automata and interface automata models with traditional scheduling analysis and enforcement algorithms [13]. Traditional scheduling analysis requires task scheduling policies and task periodicity, which are not present in a timed automata model. We exploit that extra information to calculate the response time of a given task (state) in the presence of task preemption. After the response time is obtained, it can be used to define the corresponding timing constraints in a timed automata model.

The major benefit of this approach is that it allows us to verify the real-time responsiveness of distributed systems with preemptive scheduling using timed automata models. Note that a restriction of our approach is that it assumes the scheduling algorithms used in the systems to be verified can be analyzed with an established theory.

To realize our system verification approach, we develop and formalize a new component model that supports the specification of the required component functional semantics and timing constraints as well as component composition strategies and system scheduling policies. Based on the new component model, we are developing a prototype tool to automate the process of converting our new component model into a timed automata model, after which it is possible to use an existing timed automata model checker such as UPPAAL [3] or the IF toolset [4] to verify the correctness of the model.

We now summarize features of the timed automata model, upon which our approach builds. A timed automaton[1] is a finite state Büchi automaton extended with a set of real-valued variables called *clocks*. Transitions between states are guarded by *clock constraints* which represent timing delays. More formally, let X be a set of *clock variables*. The set of *clock constraints* $C(X)$ is defined as follows:

- All inequalities of the form $x \prec c$ or $c \prec x$ are in $C(X)$, where $\prec$ is either $<$ or $\leq$ and c is a nonnegative rational number.
- If ϕ_1 and ϕ_2 are in $C(X)$, then $\phi_1 \wedge \phi_2$ is in $C(X)$.

The *timed safety automata* [11] model simplifies the timed automata model with location invariants and removes accepting locations. Formally, A *timed safety automaton* is a 6-tuple $A = (\Sigma, S, S_0, X, I, T)$ such that

- Σ is a finite set of *alphabets*.
- S is a finite set of *locations*.
- $S_0 \subseteq S$ is a set of *starting locations*.
- X is a set of clocks.
- $I : S \rightarrow C(X)$ is a mapping from locations to clock constraints, called *location invariants*.
- $T \subseteq S \times \Sigma \times C(X) \times 2^X \times S$ is a set of *transitions*. For any transition $t \in T$, $\theta_s(t)$ and $\theta_d(t) \in S$ represent the source and destination locations of a transition; $\delta(t) \in C(X)$, is the time guard which must be satisfied when the transition is taken; $\gamma(t) \in 2^X$, is a set of clocks that are reset to zero once the transition is taken.

In the subsequent sections, we extend the timed safety automata model to accommodate component abstractions and preemption semantics.

4. Component Automata

To better support modeling and verification of systems build using real-time component middleware, we have developed a new *component automaton* abstraction, which extends interface automata [6], for model specification. A component can be either *basic* or *composite*. A basic component consists of rate based *input* and *output actions* as well as a (timed) automaton which describes the behaviors of the component. The input and output actions are used to specify how a component can interact with its environment or other components. The input actions are used to model procedures/methods that can be invoked in programming languages, actions on the receiving ends of message transmission channels, or actions at the return location of a procedure/method invocation. The output actions are

used to model the invocation points of procedures/methods, the sending ends of message transmission channels as well as the point of return from a procedure/method invocation. The input and output actions that represent the return locations and return actions of procedure/method invocations, are called *returned input actions* and *returning output actions* respectively. A component automaton starts with an input action that receives requests or events at a specified rate from its environment, processes the requests, and then generates outputs to the environment again at specified rates. Figure 1 shows two examples of components, in which the transitions that are followed by "!" and "?" represent the output and input actions respectively.

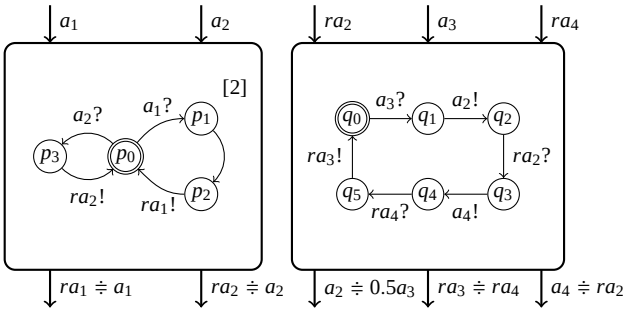

Figure 1. Two example components P and Q

In addition, a new timing constraint called a *task constraint* is used in our component abstraction. A task constraint consists of a *worst case execution time* (WCET) and a priority. The WCET represents the maximum accumulated CPU time that can be spent on a location. In the timed automaton model, the constraints over clocks do not change in accordance with automaton composition. This is usually used to represent certain cases such as timer expiration. On the other hand, WCET is also a value that can be used to calculate the response time of a state for preemptive scheduling algorithms, which we consider in Section 6.2. In Figure 1, the WCET is shown beside location p_1. The priority in a task constraint is an integer that indicates the scheduling preferences among tasks. A location with a task constraint is a *task location*; otherwise, it is a *non-task location*.

To enable scheduling analysis of components, it is also necessary to establish the relationship between the rates of input and output actions, which is a novel contribution of this work. For example, we may specify that the output rate of action a_2 for component Q in Figure 1 is half of the input rate of action a_3. There are two reasons to specify the rate relationship explicitly. First, the rate of an output action may depend on the values of certain data variables which may not be relevant to the rest of the model. Using the explicit rate specification can reduce the complexity of the model. Second, it allows us to express relationships such as the correlation between input actions more abstractly. In the rest of this paper, we will use $\div$ to denote the relationship between input and output rates as shown in Figure 1.

More specifically, a *component* $P = (\mathcal{A}_P^I, \mathcal{A}_P^O, \mathcal{A}_P^H, S_P, s_P^0, X_P, I_P, K_P, \omega_P, T_P, f_P)$ consists of the following elements

- $\mathcal{A}_P^I$ and $\mathcal{A}_P^O$ represent the input and output actions respectively. $\mathcal{A}_P^{IO} = \mathcal{A}_P^I \cup \mathcal{A}_P^O$ is the set of *external actions* of the component. $\mathcal{A}_P^I$ and $\mathcal{A}_P^O$ are mutually disjoint, i.e. $\mathcal{A}_P^I \cap \mathcal{A}_P^O = \emptyset$.
- $\mathcal{A}_P^H$ is a set of internal actions.
- S_P is a set of locations.
- $s_P^0 \in S_P$ is a starting location.
- X_P is a set of clocks.
- $I_P : S_P \to C(X_P)$ is a mapping from locations to a set of location invariants, where $C(X_P)$ is the set of clock constraints defined in Section 3.
- $K_P \subset Q^+ \times Q^+$: is a set of task constraints with WCETs and priorities, where Q^+ is the set of non-negative rational numbers. .
- $\omega_P : S_P \to K_P$ is a mapping from locations to task constraints.
- $T_P \subseteq (S_P \times A_P^{IO} \times 2^{\mathcal{A}_P^H} \times C(X_P) \times 2^{X_P} \times S_P)$ is a set of transitions.
- $f_P : \mathcal{A}_P^O \to \mathcal{F}(\mathcal{A}_P^I)$ is a function from the input actions to output rate relations.

For brevity of notation, we will use ω to represent the function from a location to its task constraint in a component; that is, if $s \in S_P$, then $\omega(s) = \omega_P(s)$. If a location s is a non-task location then $\omega(s) = \emptyset$. The disjunction operator $\vee$ for task constraints is defined as

$$\omega(s_1) \vee \omega(s_2) = \begin{cases} \emptyset & \text{if } \omega(s_1) = \omega(s_2) = \emptyset, \\ \omega(s_1) & \text{if } \omega(s_1) \neq \emptyset \text{ and } \omega(s_2) = \emptyset, \\ \omega(s_2) & \text{if } \omega(s_1) = \emptyset \text{ and } \omega(s_2) \neq \emptyset, \\ \text{undefined} & \text{if } \omega(s_1) \neq \emptyset \text{ and } \omega(s_2) \neq \emptyset. \end{cases}$$

Given a set of input actions $\mathcal{A}_P^I$, the set of output rate relations $\mathcal{F}(\mathcal{A}_P^I)$ is defined as follows :

- For all $x \in \mathcal{R}$, $x \in \mathcal{F}(\mathcal{A}_P^I)$.
- For all $x \in \mathcal{A}_P^I$, $x \in \mathcal{F}(\mathcal{A}_P^I)$.
- For all $x, y \in \mathcal{F}(\mathcal{A}_P^I)$, the expressions $x + y$, $x \times y$, $min(x, y)$ and $max(x, y)$ are all elements of $\mathcal{F}(\mathcal{A}_P^I)$.

For the convenience of future discussion, we also define the following functions which retrieve certain attributes of a transition τ in a component:

- $\theta(\tau)$ maps to a tuple (s, s') where s and s' are the source and destination locations of the transition τ respectively,
- $\alpha(\tau)$ maps to the input or output action that is associated with the transition t, and
- $\beta(\tau)$ maps to the set of internal actions that are associated with the transition t.

Given two components P and Q, the *internalized actions*, denoted as $IntA(P, Q)$, refer to the matching external actions between P and Q, i.e, $IntA(P, Q) = (\mathcal{A}_P^I \cap \mathcal{A}_Q^O) \cup (\mathcal{A}_P^O \cap \mathcal{A}_Q^I)$.

5. Component Composition

A composite component is constructed from subcomponents using a specified *component composition scheme*. There are three different composition schemes in our approach: *parallel*, *atomic*, *monitor*. Each of these schemes corresponds to a form of concurrency commonly provided by real-time component middleware: multithreaded, single threaded and cooperative multitasking respectively. The parallel composition approach is derived from the interface automata approach. The atomic and monitor composition approaches are novel contributions of our work. The parallel composition scheme cannot be used directly on a component with task locations. Section 6 discusses how to convert a component with task locations into one without them.

Formally, a composite component is defined as follows: given components P and Q, the composition of P and Q (denoted by $P \otimes Q$, $P \odot Q$ and $P \oplus Q$ for parallel, atomic and monitor composition respectively) is a component $R = (\mathcal{A}_R^I, \mathcal{A}_R^O, \mathcal{A}_R^H, S_R, s_R^0, X_R, I_R, K_R, \omega_R, T_R, f_R)$ where

- $\mathcal{A}_R^I = (\mathcal{A}_P^I \cup \mathcal{A}_Q^I) - IntA(P, Q)$, $\mathcal{A}_R^O = (A_P^O \cup \mathcal{A}_Q^O) - IntA(P, Q)$ and $\mathcal{A}_R^H = \mathcal{A}_P^H \cup \mathcal{A}_Q^H \cup IntA(P, Q)$,
- $S_R = S_P \times S_Q$,
- $s_R^0 = s_P^0 \times s_Q^0$,
- $X_R = X_P \cup X_Q$,
- $I_R : S_R \rightarrow C(X_R)$, where $I_R(s_P \times s_Q) = I_P(s_P) \wedge I_Q(s_Q)$,
- $K_R = K_P \cup K_Q$.
- $\omega_R : S_R \rightarrow K_R$ is a mapping from locations to task constraints that is defined in each composition scheme.
- $T_R \subseteq (S_R \times \mathcal{A}_R^{IO} \times 2^{\mathcal{A}_R^H} \times C(X_R) \times 2^{X_R} \times S_R)$ is subject to the composition rules for each composition scheme.
- $f_R : \mathcal{A}_R^O \rightarrow \mathcal{F}(\mathcal{A}_R^I)$.

  If a is an output action of both P and R, then the value of $f_R(a)$ is $f_P(a)$ with all internalized actions of R being recursively substituted with the values from f_Q until no internalized actions of R are in the formula. Similarly, if a is an output action of both Q and R, then the value of $f_R(a)$ is $f_Q(a)$ with all internalized actions of R being recursively substituted with the values from f_P until no internalized actions of R are in the formula.

5.1. Parallel Composition

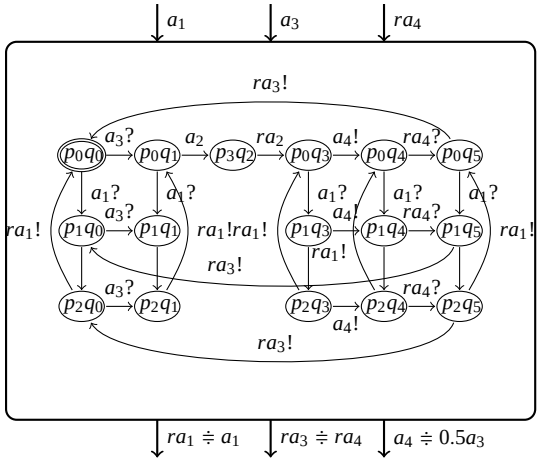

Figure 2. The composite component $P \otimes Q$

The *parallel composition* scheme is used to describe a system in which the composed components run concurrently, though the components to be composed may synchronize at the points where there are matches between the input and output actions. Figure 2 shows the parallel composition of the two components in Figure 1. In this subsection, we will only describe the case where components P and Q do not contain task constraints. We will discuss the case with task constraints in Section 6. The rules for parallel composition are defined as follows:

1. For any transition τ, where $\theta(\tau) = (s_P s_Q, s'_P s'_Q)$, $s_P \neq s'_P$ and $s_Q \neq s'_Q$, τ is a transition of R if and only if there exists a transition $\tau_P \in T_P$ where $\theta(\tau_P) = (s_P, s'_P)$ and a transition $\tau_Q \in T_Q$ where $\theta(\tau_Q) = (s_Q, s'_Q)$ such that $\alpha(\tau_P) = \alpha(\tau_Q) \in IntA(P, Q)$. The guard expression of τ is the conjunction of those of τ_P and τ_Q. The clock resets of τ are in the union of those of τ_P and τ_Q. The external actions of τ, $\alpha(\tau) = \emptyset$. The internal actions of τ, $\beta(\tau) = \beta(\tau_P) \cup \beta(\tau_Q) \cup \{\alpha(\tau_P)\}$.
2. For any transition τ, where $\theta(\tau) = (s_P s_Q, s'_P s_Q)$, τ is a transition of R if and only if there exists a transition $\tau_P \in T_P$ where $\alpha(\tau_P) \notin IntA(P, Q)$ and $\theta(\tau_P) =$

(s_P, s'_P). The actions, guard expression and clock resets of τ are the same as with τ_P.

3. For any transition τ, where $\theta(\tau) = (s_P s_Q, s_P s'_Q)$, τ is a transition of R if and only if there exists a transition $\tau_Q \in T_Q$ where $\alpha(\tau_Q) \notin IntA(P, Q)$ and $\theta(\tau_Q) = (s_Q, s'_Q)$. The actions, guard expression and clock resets of τ are the same as with τ_Q.

Rule 1 describes the synchronization between subcomponents when matches exist between input and output actions, as the actions a_2 and ra_2 shown in Figure 2. Rules 2 and 3 are a symmetric duo that describes the interleaving of actions other than those synchronization points described in rule 1. Since all component compositions are symmetric, only one of the symmetric rules for other compositions will be presented for the rest of this section.

5.2. Atomic Composition

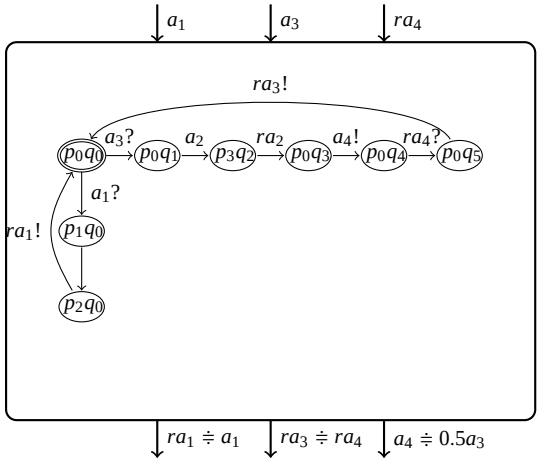

Figure 3. The composite component $P \odot Q$

Atomic composition is used to describe a system where only one subcomponent can be executed at a time, with no arbitrary interleaving between the executions of subcomponents. The interleaving can only occur when the output actions of one subcomponent match the input actions of the other subcomponents. Figure 3 shows the result of atomic composition of the two components from Figure 1 in Section 4. The rules for atomic composition are defined as follows.

1. For any transition τ, where $\theta(\tau) = (s_P s_Q, s'_P s'_Q)$, $s_P \neq s'_P$ and $s_Q \neq s'_Q$, τ is a transition of R if and only if the following conditions hold.

   - there exists a transition $\tau_P \in T_P$ where $\theta(\tau_P) = (s_P, s'_P)$ and a transition $\tau_Q \in T_Q$ where $\theta(\tau_Q) = (s_Q, s'_Q)$ such that $\alpha(\tau_P) = \alpha(\tau_Q) \in IntA(P, Q)$,
   - s_P and s_Q are not both task locations,
   - s'_P and s'_P are not both task locations.

   The guard expression for τ is the conjunction of those of τ_P and τ_Q. The clock resets of τ are in the union of those of τ_P and τ_Q. The external actions of τ, $\alpha(\tau) = \emptyset$. The internal actions of τ, $\beta(\tau) = \beta(\tau_P) \cup \beta(\tau_Q) \cup \{\alpha(\tau_P)\}$. The task constraint of $s_P s_Q$, $\omega(s_P s_Q)$, is $\omega(s_P) \vee \omega(s_Q)$; similarly, $\omega(s'_P s'_Q) = \omega(s'_P) \vee \omega(s'_Q)$.

2. For any transition τ, where $\theta(\tau) = (s_P s_Q, s_P s'_Q)$, τ is a transition of R if and only if the following conditions hold:

   - there exists a transition $\tau_Q \in T_Q$ and $\alpha(\tau_Q) \notin IntA(P, Q)$ such that $\theta(\tau_Q) = (s_Q, s'_Q)$;
   - s_P is not a task location, i.e. $\omega(s_P) = \emptyset$;
   - and one of the following conditions holds:
     - $s_P = s_P^0$,
     - there exists a transition $\tau_r \in T_R$, such that $\alpha(\tau_r) \in IntA(P, Q)$ and $\theta(\tau_r) = \left(s'_P s''_Q, s_P s_Q\right)$,
     - there exists a transition $\tau_r \in T_R$, such that $\alpha(\tau_r) \notin IntA(P, Q)$ and $\theta(\tau_r) = \left(s_P s''_Q, s_P s_Q\right)$.

   Furthermore, the actions, guard expression and clock resets of τ are the same as those of τ_Q.

Like parallel composition, rule 1 of atomic composition refers to the synchronization of input and output actions between subcomponents. However, the constraint that only one of the locations s_P and s_Q can be a task location ensures no preemption exists in atomic composition. Rule 2 enforces that transitions from different subcomponents cannot be enabled at the same time except in the initial state.

5.3. Monitor Composition

Monitor composition is used to express composition where components cooperatively share a single thread. In atomic composition, another request cannot be processed until the current one is done; however, monitor composition allows a composite component to enable an input action from one subcomponent while it is blocked on an input action from another subcomponent. For example, there exists only one execution path from (p_0q_0) and (p_0q_1) in Figure 3 before it returns to (p_0q_0), while the path diverges from (p_0q_4) in Figure 4. Notice that the divergence exists only because the transition between (p_0q_4) to (p_0q_5) is an input action from subcomponent Q which is different from the subcomponent P that provides the input action in the transition between (p_0q_4) to (p_1q_4).

Formally, the monitor composition rules are the same as those for atomic composition except for the addition of an extra condition in the third bullet of rule 2:

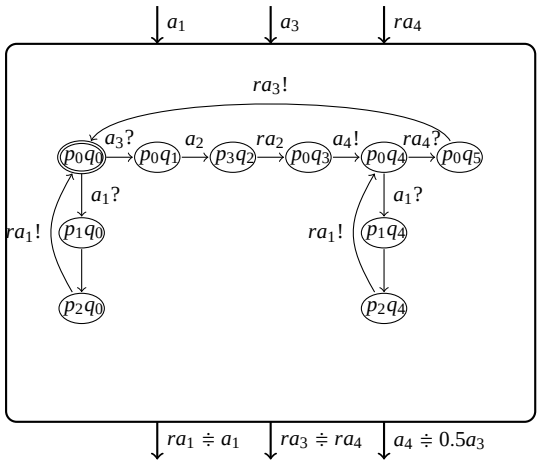

Figure 4. The composite component $P \oplus Q$

– $s_Q = s_Q^0$ and there exists a transition $\tau_R \in T_R$ such that $\theta(\tau_R) = (s_P s_Q, s'_P s_Q)$ and both $\alpha(\tau)$ and $\alpha(\tau_R)$ are input actions.

All the composition relations are symmetric (i.e., $P \bigcirc Q$ is equivalent to $Q \bigcirc P$, where $\bigcirc$ can be either $\odot$, $\oplus$ or $\otimes$). Moreover, the atomic and monitor composition relations must be nested inside parallel composition relations. For example, $P \otimes (Q \odot R)$ is legal while $P \odot (Q \otimes R)$ is not. Atomic and monitor compositions, which are designed to model component composition under single threaded and cooperative multitasking, must be used before parallel composition, which is designed to model multi-threaded composition.

6. Conversion to Timed Automata

The timed automata model does not support the modeling of preemptive systems with the specification of the maximum execution time of certain locations. The problem stems from the fact that clocks in timed automata can only progress uniformly in all locations; however, maximum execution time for a location represents the concept that time only progresses in the designated location and it should stop progressing when preemption occurs. To avoid this problem, we use response times instead of maximum execution times for model verification. However, response times generally are not available during model specification, and must be derived from the specific scheduling algorithm. For example, Table 1 shows two periodic tasks T_1 and T_2 and their expected response times when the Rate Monotonic Scheduling algorithm is used. Figure 5 shows a timed automata model of the scenario where the maximum execution time of T_2 is replaced by its respective response time. Note that the locations directlyL_1 and L_2 represent the states where tasks T_1 and T_2 are running without any other preempted tasks in the scheduler, and $L_{2,1}$ represents the state where T_1 preempts T_2 before T_2 finishes.

However, there are two problems with the model shown in Figure 5. First, the model contains a deadlock, when $t_2 > 7$ in L_2 and then transition to $L_{2,1}$. If task T_1 spends exactly 1 time units to finish, no valid transition exists because of the invariant of L_2: at that point, t_2 would be greater than 8 already and hence the transition from $L_{2,1}$ to L_2 won't be valid. Second, it is not semantically correct for T_2 to stay in L_2 for more than 5 time units without transitioning to $L_{2,1}$.

	T_1	T_2
execution time	5	1
period	20	3
response time	8	1
preemption overhead	3	0

Table 1. The parameters of T_1 and T_2

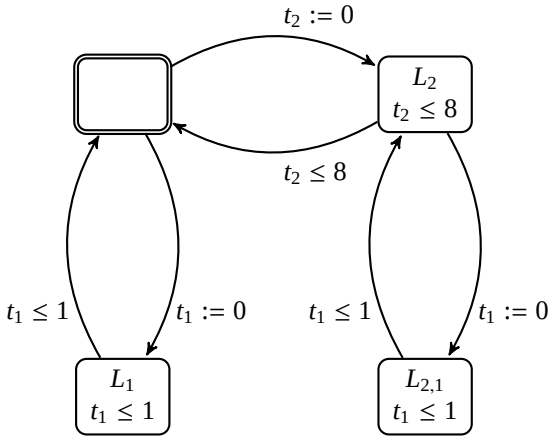

Figure 5. Timed Automata Model of T_1 and T_2 with response time transformation

6.1. Response Time with Preemption Counting Mechanism

One remedy to the deadlock problem is to add a guard $t_1 <= 7$ with the transition from L_2 to $L_{2,1}$. However, this doesn't solve the second problem mentioned above where the model allows a task to stay in a location longer than the designated maximum execution time. Without resorting to the hybrid automata model, an extra mechanism is needed to count the number of times that a task can be preempted by other tasks before its completion. In the example shown in Table 1, if Rate Monotonic Scheduling is used, T_2 can only be preempted by T_1 at most 3 times. Therefore, an integer

$C_{2,1}$ for counting the number of preemption is added into the model as shown in Figure 6.

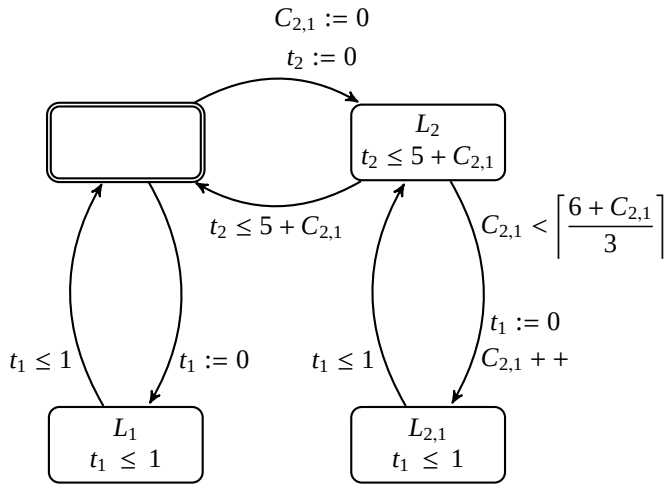

Figure 6. Timed Automata Model of T_1 and T_2 with Counting Mechanism

To be more specific, given a set of n tasks $\{T_i | 0 \leq i < n\}$ in a node, we define $J(i)$ to be the set of indexes of the tasks which have higher priority than T_i does. If the worst case execution time and periodicity of T_i are e_i and p_i respectively, then the response time r_i of task T_i can be calculated using the following formula.

$$r_i = e_i + \sum_{k \in J(i)} \left\lceil \frac{r_i}{p_k} \right\rceil e_k.$$

Our transformation utilizes the response time formula and replaces the terms $\left\lceil \frac{r_i}{p_k} \right\rceil$ with discrete counters $C_{i,k}$. These counters encode the number of times that T_i is preempted by T_k directly or indirectly. Therefore, the time t_i which the task T_i spends before completion is subject to the constraint $t_i \leq e_i + \sum_{k \in J(i)} C_{i,k} e_k$, $C_{i,k} \in \left[0, 1, \cdots, \left\lceil \frac{max(t_i)}{p_k} \right\rceil\right]$ and $max(t_i)$ is the upper bound of t_i. This constraint can be used as the invariant of the task location representing T_i and the guard for the transition that represents the termination of T_i. In addition, t_i and $C_{i,j}$ where $j \in [0, \cdots, i-1]$ are reset to zero when the task T_i starts, and the counter $C_{i,j}$ is incremented when the task T_i is directly or indirectly preempted by another task j. However, the upper bound of $C_{i,j}$ cannot be calculated directly from the formula because it depends on the upper bound of t_i and is a recursive formula. Therefore, we use the following formula to guard the transitions for which $C_{i,j}$ is incremented.

$$C_{i,j} < \left\lceil \frac{e_i + \sum_{k \in J(i)} C_{i,k} e_k + e_j}{p_j} \right\rceil$$

This guard ensures that the number of times that T_j is executed before T_i completes cannot exceed what is allowed by the specified rate of T_j.

6.2. Node Abstraction

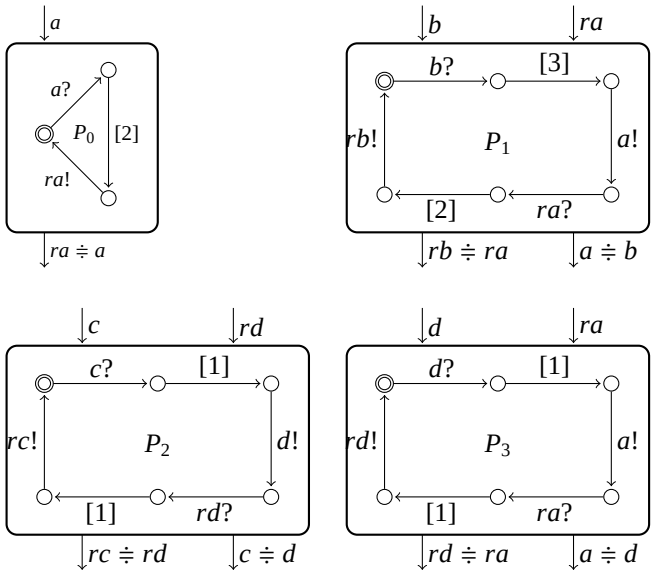

Figure 7. Example Components.

A *node* defines the boundary of a (composite) component which can be scheduled by a single processor scheduling algorithm. Given the components shown in Figure 7, we define the node N_1 to be $P_0 \otimes P_2 \otimes (P_0 \odot P_1)$. If the input actions a, b, c are periodic with frequencies 0.05 Hz, 0.02 Hz and 0.1 Hz respectively, the needed CPU utilization bound of N_1 can be easily obtained. In this case, the utilization would be 0.44, which means the node is schedulable under the Rate Monotonic Scheduling algorithm.

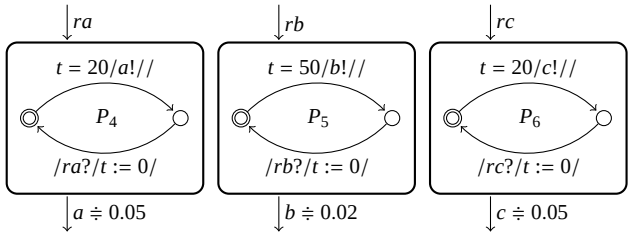

Figure 8. Example Stimulus Components.

Since a node represents a physical scheduling boundary, in addition, the atomic and monitor composition are solely used for single threaded composition. Only parallel composition can be used between components of different nodes. For convenience, brackets will be used to denote

the boundary of a node. Taking the composite component $[P_0 \otimes P_2 \otimes (P_0 \odot P_1)] \otimes [P_3]$ as an example, it contains two nodes $P_0 \otimes P_2 \otimes (P_0 \odot P_1)$ and P_3.

Similar to interface automata, a component is *open* when it contains external actions; otherwise, it is *closed*. For the components in Figures 7 and 8, the composite component $[P_0 \otimes P_2 \otimes (P_0 \odot P_1) \otimes P_0] \otimes [P_3] \otimes [P_4 \otimes P_5 \otimes P_6]$ is closed because all actions are internalized after composition.

Consider for example the node $N \equiv [P_0 \otimes P_2 \otimes (P_0 \odot P_1) \otimes P_0]$ and the composite component $M \equiv N \otimes [P_3] \otimes [P_4 \otimes P_5 \otimes P_6]$. To analyze the responsiveness of a fixed priority system, a composite component in the system has to be closed because only closed systems have enough information about the required input rates of the tasks in each node. The number of tasks in each node is the number of task locations in the node. The composite component N has 7 tasks: two from P_0, two from P_2 and three from $(P_1 \odot P_0)$. Given the output action rate relations for the components in Figure 7, it is possible to derive the input rates of all the actions in N of M. For example, the input action "b?" of $P_1 \odot P_0$ matches the output action "b!" of P_5; therefore the input rate of b in $P_1 \odot P_0$ is the same with the output rate of b in P_5, which is 0.02 Hz. Similarly, the input rates of "a?" in the two instances of P_0 in node N and "c?" in P_2 in node N are 0.05 Hz.

6.3. Task Location Conversion

Given a node which is composed of components with task locations, we define $J(i)$ to be the set of indexes of the task locations which have higher priority than a task location s_i in the node, i.e. ,

$$J(i) = \left\{ j \;\middle|\; \forall j, \text{where } s_j \text{ has higher priority than } s_i \right\}.$$

$E(i)$ represents the maximum time that can be spent on the location s_i when it is directly or indirectly preempted by s_k ($k \in J(i)$) for exactly $C_{i,k}$ time; to be more precise,

$$E(i) = e_i + \sum_{k \in J(i)} C_{i,k} e_k.$$

Before the components with task locations in a node can be composed using the rules in Section 5, the following transformation must be performed for each component:

- Identify all the task locations (tasks) in the node and sort them according their respective priorities.
- For each task location s_i, add a unique clock variable t_i and a set of counters $\left\{ C_{i,j} \;\middle|\; C_{i,j} \in \mathcal{N}, \forall j \in J(i) \right\}$ in s_i's respective component to represent the time spent on the location s_i.
- For each transition whose destination location is a task location s_i, add a clock reset $t_i := 0$ and counter resets $\left\{ C_{i,j} := 0 \;\middle|\; \forall j \in J(i) \right\}$.
- For each task location s_i, add an invariant $t_i \leq E(i)$.
- For each transition whose source location is a task location s_i, add a transition guard $t_i \leq E(i)$.

After the transformation of individual components, parallel composition within a node can be done with the rules described in section 5.1 with the addition of following rule:

4. For any transition τ, where $\theta(\tau) = (s_i s_j, s_i, s'_j)$, in addition, both s_i and s_j are task locations; then τ is a transition of R if and only if the priority of s_j is higher than that of s_i. The guard of τ is the conjunction of $C_{i,j} < \left\lceil \frac{E(i)+e_j}{P_j} \right\rceil$ and the guards of τ_i and τ_j. The actions of τ are the union of the actions of τ_j and $C_{i,j}++$.

No task constraint will remain after the above transformation and composition; therefore, we can directly use the rules in section 5.1 for the composition between nodes.

6.4. Modeling Communication Delays

Besides composition schemes, timing constraints can be added to internalized actions during component compositions. This is a unified way to specify the timing delay between component communications. The timing constraints, referred to as *composition constraints*, include a set of clock variables X' to be referred to as *composition resets* and a set of clock constraints $C(X')$ to be referred to as *composition guards*. Composition constraints may be *directed* for representing asymmetric communication overhead. For example, the composite component $P_0 \odot P_1$ from Figure 7 has two internalized actions *a* and *ra* operating in opposite directions. Thus we can use one set of composition constraints for *a* and another set for *ra*. With the previously mentioned composition schemes and the transformation of task locations, it is possible to express a variety of middleware communication and concurrency constructs rigorously and easily. Other than the wait-on-connection and wait-on-reactor communication strategies modeled by atomic and monitor composition schemes, the ACE thread pool reactor framework [16] can be modeled as parallel compositions of multiple instances of the same component automaton. Asynchronous communication channels between components can be modeled as components which provide message queue automata to be composed with event sources and sinks using the parallel composition scheme.

7. Conclusions

Real-time component based middleware helps to hide complexities from software developers; however, those hidden complexities may have an impact on the properties of

a system. These issues may be very hard to detect by developers. Significant research has been conducted to apply model checking to ease the development, assembly and verification of software systems. However, the resulting approaches do not adequately support verification of important real-time aspects of real-time component middleware-based systems.

The purpose of the research described in this paper is to provide a formal foundation for developing tools that can automatically verify properties of real-time component based systems. Our approach to modeling integrates and extends three different technologies: timed automata, interface automata and traditional schedulability analysis. Based on this research we are currently building a prototype tool for verification of real-time component based systems.

References

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

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The if toolset. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 237–267. Springer–Verlag, September 2004.

[5] S. V. A. Campos. *A quantitative approach to the formal verification of real-time systems*. PhD thesis, Carnegie-Mellon University, 1996. Chair-Edmund M. Clarke.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[7] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[8] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendor, e Sonia, and S. Yuhong. Taming heterogeneity—the ptolemy approach, 2002.

[9] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embdded software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[10] T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[11] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[12] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[13] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.

[14] G. Madl, S. Abdelwahed, and G. Karsai. Automatic verification of component-based real-time corba applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, Dec. 2004.

[15] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. *CM SIGSOFT Software Engineering Notes*, 28(5):267–276, 2003.

[16] D. C. Schmidt and C. Cleeland. Applying patterns to develop extensible and maintainable ORB middleware. *Communications of the ACM, CACM*, 40(12), 1997.

[17] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[18] V. Subramonian, C. Gill, C. Sanchez, and H. Sipma. Reusable models for timing and liveness analysis of middleware for distributed real-time embedded systems. In *6th ACM Conference on Embedded Software (EMSOFT '06)*, pages 252–261, Seoul, South Korea, Oct 2006.

[19] V. Subramonian, N. Wang, L. Shen, and C. Gill. The design and performance of configurable component middleware for distributed real-time and embedded systems. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, pages 252–261, Dec. 2004.

[20] S.V. Campos and E. Clarke. Real-Time Symbolic Model Checking for Discrete Time Models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Develpment*. World Scientific Press, AMAST Series in Computing, 1994.

[21] W. Thomas. Automata on infinite objects. pages 133–191, 1990.

[22] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker. Middleware support for aperiodic tasks in distributed real-time systems. In *RTAS '07: Proceedings of the 13th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 497–506, Washington, DC, USA, Apr. 2007. IEEE Computer Society.

[23] Y. Zhao, J. Liu, and E. Lee. A Programming Model For Time-Synchronized Distributed Real-Time Systems. In *13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '07)*, Apr. 2007.