Verification of Component-based Distributed Real-time Systems

Authors: Huang-Ming Huang and Christopher Gill

Component-based software architectures enable reuse by separating application-specific concerns into modular components that are shielded from each other and from common concerns addressed by underlying services. Even so, concerns such as invocation rates, execution latencies, deadlines, and concurrency and scheduling semantics still cross-cut component boundaries in many real-time systems. Verification of these systems therefore must consider how composition of components relates to timing, resource utilization, and other properties. However, existing approaches only address a sub-set of the concerns that must be modeled in component-based distributed real-time systems, and a new more comprehensive approach is thus needed. To address that need, this paper offers three contributions to the state of the art in verification of component-based distributed real-time systems: (1) it introduces a formal model called real-time component automata that combines and extends interface automata and timed automata models; (2) it presents new component composition operations for single-threaded and cooperative multitasking forms of concurrency; and (3) it describes how the composed models can be combined with task locations, a scheduling model, and a communication delay... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Verification of Component-based Distributed Real-time Systems

Complete Abstract:

Component-based software architectures enable reuse by separating application-specific concerns into modular components that are shielded from each other and from common concerns addressed by underlying services. Even so, concerns such as invocation rates, execution latencies, deadlines, and concurrency and scheduling semantics still cross-cut component boundaries in many real-time systems. Verification of these systems therefore must consider how composition of components relates to timing, resource utilization, and other properties. However, existing approaches only address a sub-set of the concerns that must be modeled in component-based distributed real-time systems, and a new more comprehensive approach is thus needed.

To address that need, this paper offers three contributions to the state of the art in verification of component-based distributed real-time systems: (1) it introduces a formal model called real-time component automata that combines and extends interface automata and timed automata models; (2) it presents new component composition operations for single-threaded and cooperative multitasking forms of concurrency; and (3) it describes how the composed models can be combined with task locations, a scheduling model, and a communication delay model, to generate a combined representation of the application components and supporting services that can be verified by existing model checkers. These contributions are embodied in an open-source tool prototype called the Real-time Component Model Translator (RTCMT).
Verification of Component-based Distributed Real-time Systems

Authors: Huang-Ming Huang and Christopher Gill

Corresponding Author: cdgill@cse.wustl.edu

Web Page: http://www.cse.wustl.edu/~hh1/rtcmt.html

Abstract: Component-based software architectures enable reuse by separating application-specific concerns into modular components that are shielded from each other and from common concerns addressed by underlying services. Even so, concerns such as invocation rates, execution latencies, deadlines, and concurrency and scheduling semantics still cross-cut component boundaries in many real-time systems. Verification of these systems therefore must consider how composition of components relates to timing, resource utilization, and other properties. However, existing approaches only address a sub-set of the concerns that must be modeled in component-based distributed real-time systems, and a new more comprehensive approach is thus needed.

To address that need, this paper offers three contributions to the state of the art in verification of component-based distributed real-time systems: (1) it introduces a formal model called real-time component automata that combines and extends interface automata and timed automata models; (2) it presents new component composition operations for single-threaded and cooperative multitasking forms of concurrency; and (3) it describes how the composed models can be combined with task locations, a scheduling model, and a

Notes:
This research was supported in part by NSF grant CCF-0448562 titled "CAREER: Time and Event Based
Verification of Component-based Distributed 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 software architectures enable reuse by separating application-specific concerns into modular components that are shielded from each other and from common concerns addressed by underlying services. Even so, concerns such as invocation rates, execution latencies, deadlines, and concurrency and scheduling semantics still cross-cut component boundaries in many real-time systems. Verification of these systems therefore must consider how composition of components relates to timing, resource utilization, and other properties. However, existing approaches only address a sub-set of the concerns that must be modeled in component-based distributed real-time systems, and a new more comprehensive approach is thus needed.

To address that need, this paper offers three contributions to the state of the art in verification of component-based distributed real-time systems: (1) it introduces a formal model called real-time component automata that combines and extends interface automata and timed automata models; (2) it presents new component composition operations for single-threaded and cooperative multitasking forms of concurrency; and (3) it describes how the composed models can be combined with task locations, a scheduling model, and a communication delay model, to generate a combined representation of the application components and supporting services that can be verified by existing model checkers. These contributions are embodied in an open-source tool prototype called the Real-time Component Model Translator (RTCMT).

1. Introduction

To promote the separation of application-specific and common concerns in distributed real-time systems, new forms of real-time component middleware[18, 22] support flexible configuration of timers, threads, remote communication, release guards and other common features, for each application’s needs. Unfortunately, the very flexibility that allows desirable combinations of 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 configuration that is suitable for one set of applications may introduce hazards for other applications. Specific hazards easily can be overlooked by system integrators during the component assembly process, and as an application grows larger, the expanding combinations of configuration options may make manual verification impractical.

Therefore, it is essential to develop automated tools for verification of these systems. These tools should track the compatibility of software components, provide valid middleware configuration options, and verify properties such as the absence of deadlocks or the timeliness of required responses. Model checking has emerged as an important technology for verification of distributed real-time systems in which application and middleware details can be analyzed together, but no existing approach is well suited for verification of systems built with real-time component middleware. Section 2 summarizes related work and compares our research 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 component-based distributed real-time systems. Section 3 provides an overview of our approach and a brief discussion of the timed automata model upon which it builds. This paper provides three main contributions to the state of the art in verification of component-based distributed real-time systems: (1) Section 4 introduces a formal model called real-time component automata that combines interface automata and timed automata models with task specifications; (2) Section 5 presents new real-time component composition operators for single-threaded and cooperative multitasking, and an operator for multi-threaded composition as in interface automata; and (3) Section 6 describes how composed models then can be combined with task location specifications and a scheduling model to generate a timed
automaton representation of a system with which properties can be verified by existing model checkers. Section 7 presents realistic examples that illustrate how the real-time component model can be used for verification in practice. Section 8 summarizes our contributions and offers concluding remarks.

2. Related Work

Component modeling environments: Karsai et al. [8] proposed using formal domain specific models within a software development process. In Ptolemy [7], the execution of atomic actors is described in terms of interface automata [5]. PTIDES [23] includes an executable simulation capability, but unlike our approach does not support executable composition with models of lower level middleware services. DREAM [12] and SaveCCM [4] are component-based modeling frameworks based on timed automata model checker Uppaal[3], which support model checking of tasks inside components. Unlike our approach, those models do not directly support preemptive semantics.

Compositional real-time analysis: Shin and Lee[16][17] developed a compositional real-time scheduling framework to establish global timing properties by composing timing constraints from locally analyzed tasks. A restriction of the framework is that the tasks are independent; therefore, it is not possible to analyze a system in which components may interact. The Interface Algebra[21] uses a bounded-delay resource model, the EDF scheduling algorithm and a new task workload model; there is only one scheduling component model for the entire system. Therefore, composition refers to the grouping of tasks and a task group is called a component. A limitation of this approach is that the delay and CPU capacity parameters must be assigned at the task level: if an end-to-end task consists of several sub-tasks, it requires that the CPU capacity and delay of each sub-task be determined before the Interface Algebra can be used to decide if they are compatible.

Model checking: Traditional model checkers like Spes [11] and Bogor [13] do not support explicit modeling of time. In the discrete time model, a global non-decreasing clock is maintained and monotonically incremented [20]. The discrete time model requires that continuous time be approximated by a fixed quantum, which may limit the precision with which the system is modeled. BIP[2] is a real-time component modeling framework built on top of the discrete time model. In the dense time 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, they do not express preemption semantics, since the flow conditions of the variables in a timed automata model must remain constant in all states. Hybrid automata [9] model systems where the flow conditions of variables can change among states, making it possible to represent preemption behaviors by setting the flow conditions of certain variables in some states to zero. A drawback of hybrid automata is that their verification is generally undecidable except with special constraints.

Modeling middleware services: In [19], Subramonian et al. demonstrated middleware modeling techniques that map software abstractions directly to timed automata. Although this approach epitomizes the actual implementation of software systems, it suffers from three problems: (1) models must be composed through explicit low level interactions, which is contrary to the principle of encapsulation; (2) such models express details which may not be essential for modeling the application level, and thus may inflict state space explosion [6]; and (3) unless concurrency features are encoded directly into the models[18], every software component is treated as an active object [15] which creates the potential for mismatches with different actual concurrency implementations, and makes models more difficult to develop, understand and reuse.

3. Overview of the Solution Approach

As was described in Section 1, our goal is to automate the verification of properties such as absence of deadlocks or timeliness of responses, by composing individual models of real-time software components. However, there are important limitations of existing modeling approaches: interface automata lack a way to specify and verify timing constraints; timed automata do not support preemption; model checking with hybrid automata is generally undecidable; the compositional real-time scheduling framework only works for independent tasks; and in the Interface Algebra, delay and CPU capacity must be specified before a composition can be checked. To overcome these limitations, our approach combines and extends timed automata and interface automata with a periodic workload model[16] and a fixed priority scheduling model which require knowledge of task periodicity and scheduling policies. We exploit that information to calculate the response time of each task in the presence of preemption and to define the corresponding timing constraints in a timed automata model. This approach thus allows us to verify properties of component-based distributed real-time systems with preemptive scheduling, by checking timed automata models.

To realize our verification approach, we have developed and formalized a new model called real-time component
automata that supports specification and analysis of components’ functional semantics and timing constraints, along with component composition operators and system scheduling policies. We define a node abstraction which identifies the (possibly composite) components that can be scheduled on each processor. Based on this approach, we have developed a prototype tool called the Real-time Component Model Translator (RTCMT) to automate the conversion of our new real-time component models into timed automata models, which an existing model checker can use to verify specified properties. In Sections 4, 5, and 6, we describe how the RTCMT represents and composes real-time component automata and translates them into timed automata.

**Background:** 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. Let $X$ be a set of clock variables. The set of clock constraints $C(X)$ is defined as follows: all inequalities of the form $x < c$ or $c < x$ are in $C(X)$, where $<$ is either $<$ or $\leq$ and $c$ is a non-negative rational number, and if $\phi_1$ and $\phi_2$ are in $C(X)$, then $\phi_1 \land \phi_2$ is in $C(X)$.

The timed safety automata [10] model simplifies the timed automata model with location invariants and removes accepting locations. Formally, A timed automaton is a 6-tuple $A = \langle \Sigma, S, S_0, X, I, T \rangle$ where: $\Sigma$ 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, and $T \subseteq S \times \Sigma \times C(X) \times 2^X \times S$ is a set of transitions. For any transition $t \in T$, $\delta(t)$ and $\gamma(t) \in S$ 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 with component abstractions and preemption semantics.

**4. Real-time Component Automata**

In the real-time component automata model, which also extends interface automata [5], a real-time component can be either basic or composite. A basic real-time component consists of input and output actions as well as a (timed) automaton which describes its behavior. The input and output actions are used to specify how a real-time component can interact with its environment or other components. The input actions are used to model method invocations, actions on the receiving ends of message transmission channels, or actions at the return location of a method invocation. The output actions are used to model method invocation points, the sending ends of message transmission channels, and the point of return from a method invocation. The input and output actions that represent the return locations and return actions of method invocations, are called returned input actions and returning output actions respectively. A segment of execution starts with an input action that receives requests or events from its environment, processes the requests, and then generates outputs to the environment.

**Figure 1. Two real-time components $P$ and $Q$**

Figure 1 shows two real-time components in our model, in which transition labels followed by "!" and "?" represent output and input actions respectively. A new timing constraint called a task constraint is also used, which consists of a worst case execution time (WCET) and a priority. The WCET, denoted by square brackets in our model, represents the maximum accumulated CPU time that can be spent in a location. The priority is an integer that indicates the scheduling preferences among tasks. In Figure 1, a WCET of 2 time units is shown beside location $p_1$. A location with a task constraint is a task location; otherwise, it is a non-task location. The task constraints are transformed into location invariants and transition guards based on the real-time component composition operators (which we consider in Section 5) and the preemptive scheduling algorithm used (which we consider in Section 6). A real-time component location can have a location invariant or a task constraint but not both.

More formally, a real-time component $P = (A_p, \mathcal{R}_p, \mathcal{H}_p, S_p, d_p, X_p, I_p, K_p, \omega_p, T_p)$ consists of the following elements: (i) $A_p$ and $\mathcal{R}_p$ represent the input and output actions respectively. $\mathcal{R}_p = \mathcal{R}_p^0 \cup \mathcal{R}_p^1$ is the set of external actions of the real-time component. $\mathcal{R}_p^0$ and $\mathcal{R}_p^1$ are mutually disjoint, i.e. $\mathcal{R}_p^0 \cap \mathcal{R}_p^1 = \emptyset$. (ii) $\mathcal{H}_p$ is a set of internal actions. (iii) $S_p = S_p^I \cup S_p^N$ is a set of locations, where $S_p^I$ is a set of task locations and $S_p^N$ is a set of non-task locations. $S_p^I$ and $S_p^N$ are mutually disjoint. (iv) $d_p \in S_p^N$ is a starting location. (v) $X_p$ is a set of clocks. (vi) $I_p : S_p \rightarrow C(X_p)$ is a mapping from locations to location invariants, where $C(X_p)$ is the set of clock
constraints defined. Moreover, for any $s \in S_P^T$, $I_p(s) = \emptyset$. (vii) $K_P \subseteq N \times N$: is a set of task constraints with WCETs and priorities, where $N$ is the set of natural numbers. (viii) $\omega_P : S_P \rightarrow K_P$ is a mapping from locations to task constraints. (ix) $T_P \subseteq (S_P \times A_P^T \times 2^{A_P^O} \times C(X_P) \times 2^{X_P} \times S_P)$ is a set of transitions.

If a location $s$ is a non-task location then $\omega_P(s) = \emptyset$. The disjunction operator $\lor$ for task constraints is defined as

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

For ease of discussion, we also define the following functions which retrieve attributes of a transition $\tau$ in a real-time component: $\theta(\tau)$ maps to a tuple $(s, s')$ where $s$ and $s'$ are the source and destination locations of the transition $\tau$ respectively, $\alpha(\tau)$ maps to the input or output action that is associated with the transition $\tau$, and $\beta(\tau)$ maps to the set of internal actions that are associated with the transition $\tau$. Given real-time components $P$ and $Q$, the internalized actions $IntA(P, Q)$ refer to the matched actions between $P$ and $Q$, i.e., $IntA(P, Q) = (A_P^I \cap A_Q^O) \cup (A_P^O \cap A_Q^I)$.

5. Real-time Component Composition

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

Formally, a composite real-time component is defined as follows. Given real-time components $P$ and $Q$, the composition of $P$ and $Q$ (denoted by $P \otimes Q$, $P \oplus Q$ and $P \sigma Q$ for parallel, atomic and monitor composition respectively) is a composite real-time component $R = (\mathcal{A}_R, \mathcal{R}_R, S_R, \omega_R, X_R, I_R, K_R, \omega_R, T_R)$ where:

- $\mathcal{A}_R = \mathcal{A}_P \cup \mathcal{A}_Q - IntA(P, Q)$;
- $\mathcal{R}_R = (\mathcal{A}_P^O \cup \mathcal{A}_Q^O) - IntA(P, Q)$;
- $S_R = S_P \times S_Q$;
- $s_R = (s_P, s_Q)$;
- $X_R = X_P \cup X_Q$;
- $I_R : S_R \rightarrow C(X_R)$, where $I_R(s_p \times s_Q) = I_P(s_p) \land I_Q(s_Q)$;
- $K_R = K_P \cup K_Q$;
- $\omega_R : S_R \rightarrow K_R$ is a mapping from locations to task constraints that is defined in each composition operator;
- $T_R \subseteq (S_R \times \mathcal{A}_R^T \times 2^{\mathcal{A}_R^O} \times C(X_R) \times 2^{X_R} \times S_R)$ is subject to the composition rules for each operator.

5.1. Parallel Composition

Parallel composition, denoted by operator $\otimes$, describes the case where the composed real-time components run concurrently, though they may synchronize where their input and output actions match. Figure 2 shows the parallel composition of real-time components $P$ and $Q$ from Figure 1, where $a_2$ and $ra_2$ are the only two actions that exist in both $P$ and $Q$ and thus may be synchronized in the composed automaton. Other transitions in $P$ and $Q$ can interleave arbitrarily when they are enabled at the same time.

![Figure 2. Real-time component $P \otimes Q$](image-url)
guard expression and clock resets of $\tau$ are the same as with $\tau_P$.

(3) Any transition $\tau$, where $\theta(\tau) = (s_P, s_Q, s'_P, s'_Q)$, is a transition of $R$ iff 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 $\tau$ are the same as with $\tau_Q$.

Rule 1 describes the synchronization between real-time subcomponents when matches exist between input and output actions, such as actions $a_2$ and $ra_2$ in Figure 2. Rules 2 and 3 are symmetric, describing the interleaving of actions other than those synchronization points described in rule 1. This symmetry holds for all three compositions, so only one of the symmetric rules for the other compositions will be presented.

5.2. Atomic Composition

Atomic composition, denoted by operator $\odot$, describes the case where only one real-time subcomponent can be executed at a time, with interleaving only occurring when the output actions of one real-time subcomponent match the input actions of the other. Figure 3 shows the result of atomic composition of real-time components $P$ and $Q$ from Figure 1. This composition represents the situation where a real-time component provides multiple services which must be executed sequentially rather than concurrently.

![Figure 3. Real-time component $P \odot Q$](image)

The rules for atomic composition are defined as follows:

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

- there exist both 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 $s'_Q$ are not both task locations.

The guard expression for $\tau$ is the conjunction of those of $\tau_P$ and $\tau_Q$. The clock resets of $\tau$ are the union of those of $\tau_P$ and $\tau_Q$. The external actions of $\tau$, $\alpha(\tau) = \emptyset$.

![Figure 4. Real-time component $P \odot Q$](image)

The internal actions of $\tau, \beta(\tau) = \beta(\tau_P) \cup \beta(\tau_Q) \cup \{\alpha(\tau_P)\}$. The task constraint of $s_P, s_Q, s'_P, s'_Q$ is $\omega(s_P) \cup \omega(s_Q)$; similarly, $\omega(s'_P, s'_Q) = \omega(s_P) \cup \omega(s_Q)$.

(2) For any transition $\tau$, where $\theta(\tau) = (s_P, s_Q, s'_P, s'_Q)$, $\tau$ is a transition of $R$ iff 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 provisions holds:
  - (i) $s_P = s'_P$,
  - (ii) there exists a transition $\tau_r \in T_R$, such that $\alpha(\tau_r) \in IntA(P, Q)$ and $\theta(\tau_r) = (s_P, s'_Q, s_P, s'_Q)$, or
  - (iii) there exists a transition $\tau_r \in T_R$, such that $\alpha(\tau_r) \notin IntA(P, Q)$ and $\theta(\tau_r) = (s_P, s'_Q, s_P, s'_Q)$.

Furthermore, the actions, guard expression and clock resets of $\tau$ are the same as those of $\tau_Q$.

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

5.3. Monitor Composition

Monitor composition, denoted by operator $\oplus$, describes the case where real-time components cooperatively share a single thread. In atomic composition, another request cannot be processed until the current one is completed; however, monitor composition allows a composite real-time component to enable an input action from one real-time subcomponent while it is blocked on an input action from another. For example, in Figure 3 there is only one execution path from $(p_0q_1)$ to $(p_0q_0)$, while the path diverges at $(p_0q_2)$ in Figure 4, which illustrates monitor composition of real-time components $P$ and $Q$ from Figure 1. The divergence
exists only because the transition from \((p_0q_4)\) to \((p_0q_5)\) is on an input action from \(Q\) whereas \(P\) provides the input action in the transition from \((p_0q_4)\) to \((p_1q_4)\). The monitor composition rules are the same as for atomic composition, except for a relaxation of the third condition of rule 2 by adding the provision: (iv) \(s_Q^\theta = s_Q^0\) and there exists a transition \(\tau_R \in T_R\) such that \(\theta(\tau_R) = (s_P^\theta s_Q^0 s_P^\theta s_Q)\) and both \(a(\tau)\) and \(a(\tau_R)\) are input actions.

5.4. Node Boundaries and Operator Precedence

A node specification is also needed to enable real-time analysis for distributed and multi-core systems. A node defines the extent of a (possibly composite) real-time component which uses a single processor. We denote node boundaries with curved braces in a composition expression, e.g., \(\{P \otimes Q\} \otimes \{R\}\).

The composition operators in our approach represent the different concurrency strategies used in modern middleware frameworks. Atomic composition is primarily used to connect real-time components via method calls or service handlers. Monitor composition is used for connecting real-time components via cooperative multitasking. Parallel composition within a node connects real-time components via preemptive multitasking. Parallel composition of nodes (i.e., in a distributed or multi-core system) constitutes non-preemptive (physically parallel) multitasking. Since a node represents a physical scheduling boundary, atomic and monitor composition are solely used for real-time components within a node, and only parallel composition can be used between real-time components on different nodes.

A natural operator precedence order, which our real-time component model enforces, arises from the definitions of the composition operators and the node boundaries. Atomic composition is only defined over real-time components that execute completely before yielding the single thread to another real-time component, and thus has highest precedence. Monitor composition still assumes single-threaded execution and thus has second highest precedence. Parallel composition within a node has third highest precedence since it allows arbitrary concurrency of its real-time subcomponents but depends on a common processor within that node. Parallel composition between nodes has lowest precedence.

6. Conversion to Timed Automata

In this section we describe how our real-time component model can be converted by the RTCMT tool into an equivalent timed automata representation for verification with an existing timed model checker. An important challenge in achieving this conversion is that timed automata do not easily support the modeling of preemptive real-time systems. The problem stems from the fact that clocks in timed automata can only progress uniformly in all locations even though preemption assumes that time progresses in a designated location and it should stop progressing there when preemption occurs. To overcome 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 for a specific scheduling algorithm. For example, consider tasks \(T_1\) and \(T_2\) which have periods of 3 and 20 time units, and WCETs of 1 and 5 respectively, under rate monotonic scheduling.

![Figure 5. Timed automata model of \(T_1\) and \(T_2\) with response time transformation](image)

To illustrate the complexities that must be addressed, 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 locations \(L_1\) and \(L_2\) represent states where tasks \(T_1\) and \(T_2\) are running without any other tasks in the scheduler, and \(L_{2,1}\) represents the state where \(T_1\) preempts \(T_2\) before \(T_2\) finishes. The text inside a directed edge is a 3-tuple, separated by delimiter /, representing the attributes of a transition if present. The first and second elements of the tuple give the guard and the external actions, while the third element gives the internal actions and/or clock resets of the transition.

There are two problems with the model shown in Figure 5, which we address in this section. First, the model deadlocks when \(t_2 > 7\) in \(L_2\) and then a transition is taken to \(L_{2,1}\). If task \(T_1\) spends exactly 1 time unit to finish, no valid transition exists because of the invariant of \(L_{2,1}\): at that point, \(t_2\) already would be greater than 8, 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 \(L_{2,1}\). These problems motivate the following refinements to our approach.

**Preemption counting:** Our solution to the deadlock problem is to add extra counters to the model in order to count the number of times that a task can be preempted by other tasks before its completion. For the previous example, we introduce a variable \(C_{2,1}\) to represent the number of times \(T_2\) is preempted by \(T_1\). As Figure 6 illustrates, \(C_{2,1}\) is...
incremented when \( T_2 \) is preempted by \( T_1 \), and the invariant of \( L_2 \) is changed to \( t_2 \leq 5 + C_{2,1} \) which represents the response time of \( T_2 \) when \( T_2 \) is preempted by \( T_1 \) exactly \( C_{2,1} \) times. We define \( HP(i) \) to be the set of indexes of the task locations which have higher priority than task \( i \) in location \( s_i \); \( e_i \) to be the WCET of task \( i \); and \( C_{j,k} \) to be the number of times that task \( j \) can be directly or indirectly preempted by task \( k \). The maximum time that can be spent in location \( s_i \) is \( e_i + \sum_{c \in HP(i)} C_{i,c} e_c \). In addition, we use separate automata to output task start events periodically. In Figure 6, real-time components \( \Gamma_0(a, 3) \) and \( \Gamma_0(b, 20) \) (which instantiate the real-time component template\(^2\) in Figure 7 with different parameters) trigger the transitions in \( T_1 \) and \( T_2 \) with corresponding periodicities. The transition from \( L_2 \) to \( L_{2,1} \) is thus subject to the timing constraints specified in \( \Gamma_0(a, 3) \) and \( \Gamma_0(b, 20) \), without needing to specify an upper bound on \( C_{2,1} \).

Under-constrained and over-constrained models:
Even with those transformations, the resulting timed automata still contain some behaviors that couldn’t possibly happen in a real systems. Consider Figure 6 without the underlined constraints. A trace like \( L_0 \overset{a_1}{\rightarrow} L_2 \overset{a_2}{\rightarrow} L_{2,1} \overset{a_4}{\rightarrow} L_2 \overset{a_5}{\rightarrow} L_2 \overset{a_6}{\rightarrow} L_0 \) would be allowed in the model, but it couldn’t happen in a real system because the trace stays in \( L_2 \) for more than 5 time units which would exceed the maximum execution of \( T_2 \). We call this kind of transformed model under-constrained. One remedy to this problem is to strengthen the constraints with transition guards such that the transitions out of task locations can only be taken at exactly the corresponding WCET time units, as shown in Figure 6 by the underlined constraints. We call this kind of model over-constrained because not all behaviors that could happen in the system are represented in the model. For example, the case where task \( T_1 \) finishes in 0.5 time units is not represented by the over-constrained model in Figure 6.

Although our transformations thus cannot model preemptive systems in perfect fidelity, the over- and under-constrained models are still very useful to check the properties of a system. The under-constrained model can be used to check if certain desired properties will be eventually/globally true for all traces of a system, because an under-constrained model covers all behaviors of the real systems. The over-constrained model is useful to find (more rapidly) traces that contain undesired properties such as a deadlock or a timing violation and to track down the sources of problems, since any problems found using an over-constrained model also exist in the system.

**Urgency:** All input and output actions in our real-time component model are synchronous; i.e., a transition with internalized actions won’t be taken until all the guards on the transition are enabled. We adopt the urgent semantics used for the urgent channels in UpPAAAL, for all actions in our model; i.e., a transition with internalized actions will be taken without delay as soon as it becomes enabled.

Taking the real-time component template \( \Gamma_1 \) in Figure 8 as an example, if action \( a \) was not treated as urgent, the system could stay in the starting location forever, even if \( a \) was enabled in other real-time subcomponents. To ensure the action is eventually taken without relying on urgent semantics, an invariant \( t \leq \pi \) would be required for the starting location of \( \Gamma_1 \). However, it is often impractical for a system designer to anticipate the maximum queuing delays for I/O actions without knowledge of the entire system. As a con-

\[ \text{Figure 6. Timed automata model of } T_1 \text{ and } T_2 \text{ with preemption counting mechanism} \]

\[ \text{Figure 7. Real-time component template } \Gamma_0 \]

\[ \text{Figure 8. Real-time component template } \Gamma_1 \]

\[ ^2\text{For compactness of representation, we adapt the parametrized model template approach from UpPAAAL to our real-time component models.} \]
sequence, we choose to use urgent semantics exclusively in our real-time component model.

**Communication delays:** Our real-time component model also allows explicit specification of timing delays, e.g., in real-time component communications. The process of adding a delay \( \delta \) to a transition \( \tau \) from location \( L_0 \) to location \( L_1 \) involves replacing \( \tau \) in the model with (1) a new location \( L_2 \) with an invariant \( t \leq \delta \); (2) a new transition from \( L_0 \) to \( L_2 \) with a clock reset \( t := 0 \); and (3) a new transition from \( L_2 \) to \( L_1 \) with a clock reset \( t := 0 \).

### 7. Illustrative Verification Examples

With the previously mentioned real-time composition operators and the transformation of task locations and transitions, it is possible to express a variety of middleware communication and concurrency constructs rigorously and easily. The `WaitOnConnection` and `WaitOnReactor` strategies (where remote method calls are handled in a blocking or non-blocking manner, respectively [18]) are modeled directly by the atomic and monitor composition operators respectively. A thread pool framework [14] can be modeled as parallel compositions of multiple instances of the same real-time component automaton. Asynchronous communication channels can be modeled as real-time components which provide message queue automata to be composed with event sources and sinks using the parallel composition operator.

With the ability to analyze systems with dependent tasks, it is also fairly easy to model critical sections protected by semaphores using a priority ceiling protocol in our framework. If a task contains a critical section, it can be divided into a sequence of sub-tasks separated by the critical sections where the critical sections are also modeled as sub-tasks. All tasks except the critical sections will assume the priority of the original task. These sub-tasks are then connected by transitions according to their execution order. Critical section sub-tasks guarded by the same semaphore in a node should all be assigned the same priority, whose value is greater than that of any of the original tasks from which the sub-tasks were obtained. Since the critical sections have higher priorities than the related original tasks, they won’t be preempted by those tasks in the model. We now present two more comprehensive examples to illustrate how our real-time component model can be used to verify the properties of real world systems.

#### 7.1. Verification with Concurrency and Priority Effects

The first example, in which the constituent real-time components are instantiated from the templates shown in Figure 9, is shown in Figure 10. This example is based on [18] and it demonstrates how properties of a component-based distributed real-time system can be affected by the choice of concurrency strategies used by underlying middleware. It consists of 3 nodes: `Client`, `S1` and `S2`. Real-time component \( P_0 \) in node `Client` initiates output action \( a \) within 3 time units and real-time component \( P_1 \) in `S1` waits for action \( a \), processes it for 25 time units and then relays it to real-time component \( P_2 \) in `S2` for further processing. Similarly, \( P_3 \) waits for input action \( b \) from \( P_1 \), processes it for 25 time units and then relays it to \( P_2 \) in `S1`. When \( P_2 \) completes processing in another 25 time units, it issues action \( rc \) and returns to its initial state. Subsequently, \( P_3 \) and \( P_1 \) will return to their initial states when the transitions with actions \( rc \) and \( rb \) are enabled. If the transition with action \( ra \) in \( P_0 \) is taken within the deadline of 100 time units, the initial location in \( P_0 \) will be reached; otherwise a `deadline miss` location will be reached.
Since node $S1$ consists of two real-time components, $P_1$ and $P_2$, different composition operators could be used. We transformed the models with various composition operators into timed automata models and verified the transformed models with the UPPaal model checker, using the temporal logic expression $E<>$ deadlock to see if there was a deadlock in the over-constrained model. For the case with atomic composition $P_1 \odot P_2$, which modeled two CORBA services configured with a single thread and a WaitOnConnection strategy, (or a co-location optimization as in TAO [14]) the model checker successfully detected and showed a trace that led to deadlock. Under atomic composition, the transitions with input action $c$ in $P_2$ and output action $c$ in $P_3$ were not simultaneously enabled, and thus the system reached a deadlock.

We also used the expression $E<>$ deadlock to do a quick check for the existence of deadlock in the over-constrained models for the cases $SI = \{P_1 \oplus P_2\}$ and $SI = \{P_1 \otimes P_2\}$, which represented that node $SI$ was configured with a single-threaded WaitOnReactor strategy or a multi-threaded concurrency strategy, respectively. The model checker indicated that the property was not satisfied in either case. We then used the expression $A[\]$ !deadlock to check the under-constrained models and the model checker reported the property was satisfied, at which point we were sure there was no deadlock in either of those two cases. Similarly, the model checker also reported no deadline miss when we used $A[\]$ !Client.deadline_miss to check the under-constrained models. However, if another node $Client_2$ with real-time component $P_4 \equiv \Gamma_4(d, 100)$ was added to the system and node $SI$ added real-time component $P_5 \equiv \Gamma_2(d, 25)$ to accept the input action from $P_4$, a deadline miss could still occur no matter whether $SI$ had monitor or parallel composition. The resulting traces showed that the deadline miss happened when the transition with action $d$ is taken immediately before both transitions with action $c$ in $P_2$ and $P_3$ were enabled. If we refined the system to use parallel composition but with priorities assigned so that $T_3$ of $P_1$ and $T_0$ of $P_2$ had higher priorities than $T_0$ of $P_3$, which modeled a multi-threaded system with different priority lanes, then the deadline wouldn’t be missed in the resulting system.

### 7.2. Verification with Priority, Delay, and Deadline Effects

Figure 11 shows a system with two periodic message processing flows, which in addition to $\Gamma_1$ and $\Gamma_4$ also instantiates real-time component template $\Gamma_5$ shown in Figure 12. The first flow is generated by node $RateGen1$ with a period of 10 time units, is processed by task $T_1$ of $P_2$ and subsequently by tasks $T_0$ of $P_3$ and $T_2$ of $P_2$. The second flow is generated by $RateGen2$ and is only processed by task $T_0$ of $P_3$. $P_2$ and $P_3$ are collocated in the same node; therefore, they are subject to mutual interference through preemptive scheduling. We assign tasks in $P_2$ to have higher priorities than those in $P_1$ according to rate monotonic scheduling.

An important part of this model is the real-time components $P_0$ and $P_1$ on the $RateGen1$ and $RateGen2$ nodes, which enable the transitions with output actions $a$ and $c$ in the interval of 10 and 40 time units, respectively. If those real-time components fail to receive responses within their deadlines (represented by the $\delta$ variable in $\Gamma_4$), the deadline miss location will be reached. Therefore, we used the temporal logic expression $A[\] ! (RateGen1.deadline_miss || RateGen2.deadline_miss)$ to check whether the system was schedulable. With the under-constrained model transformed from the example in Figure 11, the UPPaal model checker could verify it was schedulable because the above temporal logic expression was satisfied in all executions of the model. We then changed the model to impose communication delay of 2 time units between $SI$ and $S2$, and in another trial shortened the deadline of $RateGen2$ to 9, and in subsequent verification with UPPaal, the temporal logic expression was not satisfied in either of those cases. We also obtained a deadline miss trace (by checking $E<> RateGen1.deadline_miss || RateGen2.deadline_miss$ with the over-constrained model) in each trial. Therefore, that the system would not be schedulable with either of those modifications was easily detected using the under-constrained models, and the sources of the problems were easily identified using the over-constrained models.
8. Conclusions

Real-time component middleware helps to hide complexities from software developers; however, those hidden complexities may have an impact on crucial properties of a system, which may be very hard to detect without automatic verification tools. Significant research has been conducted to apply model checking to ease the development, assembly and verification of software systems. However, existing approaches do not adequately support verification of component-based distributed real-time systems.

The research presented in this paper provides a formal and practical foundation for automatic verification of properties of component-based distributed real-time systems. Our approach to modeling these systems integrates and extends: timed automata, interface automata and traditional schedulability analysis. The RTCMT tool introduced in Section 3 and the illustrative examples presented in Section 7 are available for download as open-source software at www.cse.wustl.edu/"hh1/rtcmt.html.

References