Toward Real-Time Component-based Systems

Shengquan Wang, Sangig Rho, Riccardo Bettati, and Wei Zhao

***Abstract*— Component technology has become a central focus of software engineering in research and development. Reusability is a key factor that contributes to its success. The reuse of components can lead to a shortening of software development cycles and savings in software development costs. However, existing component models provide no support for real-time services and some real-time extensions of component models lack of consideration for reusability of components in providing both functional and real-time services. In this work, we develop a real-time component-based system that enables true reusability of both functional and real-time services.**

I. Introduction

Component technology has become a central focus of software engineering in research and development due to its great success in market. Reusability is a key factor that contributes to this success [1]. With component technology, software systems are built by assembling components that have already been developed earlier, with integration in mind. With software component frameworks, the non-functional code is automatically generated, and system developers can focus on core business logic parts, without wasting time with common non-functional parts. The reuse of components and developers' focusing on core parts lead to a shortening of software development cycles and savings in software development costs.

Although component-based models deal successfully with functional attributes, they provide little support for real-time services. Existing standards – such as CORBA, COM+, and EJB – are unsuitable for real-time applications because they do not address issues of timeliness and predictability of service, which is basically required by real-time systems [2]. OMG working group has proposed a specification for a real-time CORBA [3], [4]. However, there is no specification for a real-time EJB or a real-time COM+ yet. The TAO project [5] provided a CORBA implementation that guarantees that calls across components preserve priority levels and that the overhead in servicing a call request is statically predictable. The VEST toolkit [6] provided a rich set of dependency checks based on the concept of aspects to support distributed embedded system development via components.

Most of these real-time extensions use traditional approaches to provide real-time service guarantees: Real-time services are typically provided in form of descriptions of the execution time, period, priority, and deadline to meet expectations for each method invocation. Applications would need intricate knowledge of the underlying hardware architecture and system software (such as subroutine procedures) in the target environment to estimate these parameters accurately. This is a big burden for any application, especially in large-scale and heterogeneous environments. Moreover, with these traditional approaches in component-based systems, components may not be reusable any more in terms of providing real-time services.

Our goal in this work is to develop a real-time component-based system that enables true reusability of both functional and *real-time* services. To achieve this, we add a real-time service specification to each component to make it become what we call a "real-time component". We build a component-based resource overlay to isolate the underlying resource management from applications. This resource overlay will make real-time components be truly reusable and also separate the responsibilities among application designers and components providers.

To the best of our knowledge, ours is the first work to identify and address the reusability of components in terms of providing both functional and real-time services.

II. Component-based Resource Overlays

In component software, a component has three basic characteristic properties [1]: (i) *Isolation* – A component should be deployable independently as an isolated part: Neither the environment nor other components or a third party have access to its construction details. The component is also an atomic unit of deployment, as it will never be deployed partially. (ii) *Composability* – A component should be composable with other components. It needs to be a self-contained function unit with well-specified interfaces. A third party can access the component through the contractually specified interfaces. (iii) *Opaqueness* – A component has no (externally) observable state. Fig. 1 illustrates a component architecture.

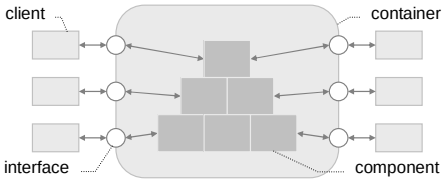

Fig. 1. An illustration of a component architecture

We extend the component architecture described above to build a *real-time* component architecture. For this, we augment the largely functional interfaces and context dependencies with *contractually specified temporal interfaces* and *explicit time-related context dependencies*. Any such augmentation of the component interface architecture should continue to satisfy the three basic component properties described earlier: (i) The real-time interface architecture should not interfere with the isolation property. Each component should be separated from other components in providing real-time service guarantees. For example, uncontrolled resource conflicts among different components should be avoided. (ii) Composability should be maintained. The real-time service interface should well

The authors are with the Department of Computer Science, Texas A&M University, College Station, TX 77843. Email: {swang, sangigr, bettati, zhao}@cs.tamu.edu.

represent the real-time service provided by the component. Applications can access the service through the real-time service interface. (iii) The real-time interface architecture should maintain opaqueness. Applications do not need to know how real-time services are provided by each component. The interfaces should not include information relating to the underlying component implementation, such as methods' worst-case execution time, scheduling algorithm used in method execution in components, for example.

We use a very simple contractual interface, which formulates the real-time service provided in terms of the service guarantee (described in form of a deadline) given a worst-case arrival (described in form of an arrival function). We first introduce the arrival function.

Definition 1 (Arrival function): If the maximum number of method invocations during any time interval of length I is bounded by $\mathcal{A}(I)$, we define $\mathcal{A}$ as an arrival function of this sequence of method invocations. For example, a bursty arrival can be described using a burst size σ and average arrival rate ρ as $\mathcal{A}(I) = \sigma + \rho I$.

The arrival function $\mathcal{A}$ and the deadline D give a contractual definition of the real-time service provided by the component: For any sequence of invocations, if its arrival function is below $\mathcal{A}$, this component will guarantee that any invocation in this sequence will meet its deadline D at this component.

This interface specification clearly meets the isolation, composability, and opaqueness requirements for real-time components. In order to let components provide more flexible service and better utilize the underlying resource usage, we extend the above specification by introducing different service levels and taking into consideration different methods exposed by components. We define *class of service* as the service level for each component. Assuming there are M classes of service, we define class-i real-time service for Component e as $\langle \Theta_{e,i}, \mathcal{A}_{e,i}, D_{e,i} \rangle$, where $\Theta_{e,i}$ is a group of methods exposed by Component e, $\mathcal{A}_{e,i}$ is an upper-bound on arrival function of invocations of method in $\Theta_{e,i}$, and $D_{e,i}$ is a deadline for any invocation of method in $\Theta_{e,i}$. In other words, for a sequence of invocations of methods in $\Theta_{e,i}$ with arrival function $\mathcal{A}_{e,i}$, Component e can guarantee a worst-cased delay bounded by $D_{e,i}$ for any method invocation in this sequence. For example, assume that Component e exposes four methods $\theta_1, \cdots, \theta_4$ and defines four classes, an real-time service interface specification is illustrated in Table I.

TABLE I

AN ILLUSTRATION OF A REAL-TIME SERVICE INTERFACE SPECIFICATION

class i	$\Theta_{e,i}$	$\mathcal{A}_{e,i}(I)$	$D_{e,i}$
1	θ_1, θ_2	$1 + 2\ I$	0.050 sec
2	θ_1, θ_2	$2 + 8\ I$	0.250 sec
3	$\theta_1, \theta_2, \theta_3, \theta_4$	$3 + 9\ I$	0.150 sec
4	θ_3, θ_4	$1 + 4\ I$	0.300 sec

Based on real-time components, we build a resource overlay, which aims to isolate the underlying resource management from applications. The resource overlay is composed of a bunch of resource overlay nodes – real-time components. In the resource overlay, each real-time component is a real-time service resource unit – an abstraction of resource – besides a function unit. From the applications' point of view, the resource unit is real-time components instead of the underlying resource (such as CPU, memory). Component providers will take care of the mapping of component-based overlay resource to the underlying resource while providing real-time services to applications through the real-time service interface. Fig. 2

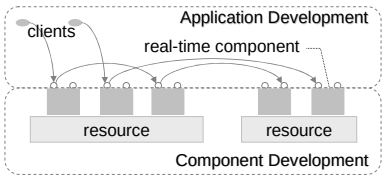

Fig. 2. An illustration of a component-based resource overlay

illustrates a component-based resource overlay. The resource overlay separates component development from application development and makes the underlying resource management totally transparent to application designers. The real-time service interface specification in real-time components does not touch any information of their underlying implementation. Any change in the implementation of components will not affect the application design.

III. BUILDING REAL-TIME COMPONENTS

It is component providers' responsibility to implement the component functionality in the form of a set of interfaces $\langle \Theta_{e,i}, \mathcal{A}_{e,i}, D_{e,i} \rangle$'s. Service implementation issues can be divided into two categories: (i) Inter-component – Recall that each real-time component should be isolated from others in terms of the underlying resource usage to meet the isolation requirement. The underlying resource could be CPU or memory (here we focus on CPU). We use a guaranteed-rate scheduler to ensure temporal isolation of components on the same processor and allocate required processor utilization to each component. A Total Bandwidth Server [7] can achieve this; (ii) Intra-component – Each component will provide multiple classes of service. To differentiate among classes of service in the same component, we use a simple static-priority scheduler, and use the class-id as priority level.

The only unsolved implementation issue is to how to determine the processor utilization assigned to each component. We will address this in the rest of this section.

Since the component implementation is bound to the underlying hardware platform, the execution of the component's methods can be easily characterized at component-implementation time. In particular, each exposed method can be associated with its worst-case execution time (WCET) on the specific platform. We aim to compute the worst-case delay suffered by execution of any method in $\Theta_{e,i}$. For this, we denote $C_{e,i}$ as the maximum WCET of all methods in $\Theta_{e,i}$. In conjunction with the arrival function defined as part of the real-time service interface specification, the WCET gives rise to the workload characterization for the method set $\Theta_{e,i}$ on the underlying implementation platform.

Definition 2 (Workload function): If the cumulated execution time of a sequence of method executions is bounded by

$\mathcal{F}(I)$ during any time interval with length I, we define $\mathcal{F}$ as a workload function of this sequence of method executions.

Given the invocation arrival function $\mathcal{A}_{e,i}(I) = \sigma_{e,i} + \rho_{e,i}I$ for Component e of Class i and the associated $C_{e,i}$ of $\Theta_{e,i}$, the workload function for Component e of Class i can be expressed as $\mathcal{F}_{e,i}(I) = C_{e,i}\mathcal{A}_{e,i}(I)$. If we assume a constant processor utilization α_e to be assigned to real-time Component e, we can use a time demand/supply argument [7] to derive the worst-case delay $d_{e,i}$ suffered by any method invocation in Component e of Class i as follows:

$$d_{e,i} \leq \frac{\sum_{p \leq i} C_{e,p}\sigma_{e,p}}{\alpha_e - \sum_{p<i} C_{e,p}\rho_{e,p}} \leq D_{e,i}. \quad (1)$$

In order to satisfy all classes of service, by (1), the allocated processor utilization for components has to be set at least as

$$\alpha_e = \max_{1 \leq i \leq M} \{\frac{1}{D_{e,i}} \sum\nolimits_{p \leq i} C_{e,p}\sigma_{e,p} + \sum\nolimits_{p<i} C_{e,p}\rho_{e,p}\}. \quad (2)$$

When allocating processor utilization to components, component developers should ensure that the overall processor utilization does not exceed the safe utilization level allowed by the specific platform.

IV. Building Real-Time Applications

In our system, applications include *clients* and *application servers* where a bunch of components have been deployed in. Each invocation from a client can cascadedly trigger execution of one or more methods, either on a single component or on several components. These components in turn can be located on one or across several application servers. A sequence of client invocations resulting in a sequence of cascaded method execution is called a *task* (Fig. 3). In a component-based

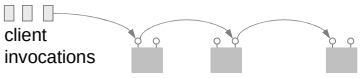

Fig. 3. Task Model

system, an invocation from a client can pass through several components and we assume all invocations in the same task to execute on the same components in the same order. Tasks in applications can be modeled as a *task graph* (Fig. 4), which is, by its nature, a directed acyclic graph. Each node is a

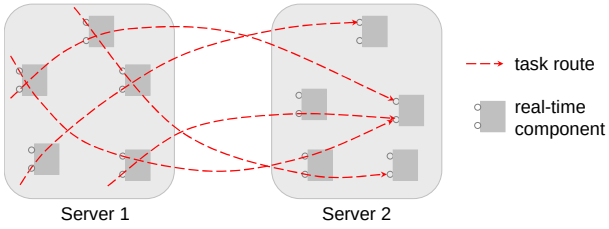

Fig. 4. Task graph

component and each task forms a *task route*. In a task route, a task at a component has a certain set of output, which is a set of inputs for the task at next component along the task route. There could be multiple tasks along each task route.

Any task is associated with a end-to-end deadline requirement and a source arrival function. To provide real-time service guarantees, application designers have to ensure any invocation in a task meets the end-to-end deadline requirement. Moreover, each task will consume the resource in the resource overlay. The application designer must ensure that the real-time service specified in each real-time component will not be violated. Therefore, an admission control mechanism has to be in place. For an admission request of Task T, there are two issues in admission control procedures that we have to address in building real-time applications:

First, what is the worst-case end-to-end delay experienced by any invocation in Task T? In Task T, all of its invocations have an end-to-end deadline D^T requirement. Assume each invocation in Task T will go through sequence of components e_h of class i_h, $h = 1, 2, \ldots, H$, and any method Task T will call in Component e is in $\Theta_{e,i}$. Recall that the worse-case delay provided by Component e of Class i is $D_{e,i}$. To guarantee the end-to-end deadline for any invocation in Task T, application designers have to ensure that the end-to-end delay d^T suffered by any client invocation in Task T should be bounded as

$$d^T = D_{e_1,i_1} + \cdots + D_{e_H,i_H} \leq D^T. \quad (3)$$

Second, what is the consumed resource by Task T? Provided that a task has an arrival function $\mathcal{A}^{in}(I)$ before arriving at a component, the arrival function will become $\mathcal{A}^{out}(I) \leq \mathcal{A}^{in}(I + d)$ just after a worst-case delay d at this component. We define $\mathcal{A}^T$ as the source arrival function of Task T (before calling the first component). If $\mathcal{A}^T(I) = \sigma^T + \rho^T I$, then the consumed resource by Task T at Component e of Class i is

$$\mathcal{A}^T_{e_h,i_h}(I) = (\sigma^T + \rho^T D_{e_1,i_1} + \cdots + \rho^T D_{e_{h-1},i_{h-1}}) + \rho^T I. \quad (4)$$

The application designer must ensure that the real-time service specified in each real-time component along the task route Task T will go through will not be violated, i.e.,

$$\sum\nolimits_{T' \in S_{e_h,i_h}} \mathcal{A}^{T'}_{e_h,i_h}(I) \leq \mathcal{A}_{e_h,i_h}(I), \quad (5)$$

where S_{e_h,i_h} is the set of existing tasks that use class-i_h service of Component e_h.

Due to the space limitation, the details of the admission control procedures will not be described here.

V. Performance Evaluation

We used Enterprise JavaBeans (EJB) [8] as the underlying framework for the realization of a real-time component-based system. The implementation is based on JBoss [9] (Version 3.2.1), which is a popular, free, open source Java 2 Platform, Enterprise Edition (J2EE, including EJB) implementation. We adopt TimeSys Linux RT 3.1 for our real-time operating system [10]. We add real-time RMI into the Reference Implementation of Real-Time Specification for Java (RTSJ-RI) from TimeSys. Based on the real-time infrastructure, we built a real-time component-based system, where *admission control* is one of main modules.

As we know that one of the basic features in our designed systems is the reusability of real-time components. However,

this feature cannot be evaluated quantitatively. Instead, we will measure the performance in terms of real-time metrics: the delay for each invocation from clients and the admission probability with admission control.

In our experiments, we assume CPUs in EJB servers are the bottleneck. We choose two Pentium III machines with 933 MHz CPU as clients, one Pentium 4 machine with 2.53 GHz CPU as an application server. These three machines are in the same subnet. The application server is installed with real-time component-based systems software and deployed with two real-time components $\{e_1, e_2\}$. Component e_j will expose one method θ_{e_j} and define a single class of service (therefore we can ignore the class index), and its real-time service interface is $\langle \Theta_{e_j}, \mathcal{A}_{e_j}, D_{e_j} \rangle$, where $\Theta_{e_j} = \{\theta_{e_j}\}$, $\mathcal{A}_{e_j}(I) = 2.5 + 2.0I$ and $D_{e_j} = 1.250$ sec, for $j = 1, 2$. Each component will be allocated $\alpha_{e_j} = 40\%$ processor utilization and the method exposed by each component is associated with a WCET $C_{e_j} = 0.226$ sec. The runtime method execution will be assigned a single real-time priority.

a) Our system with enabled admission control vs. with disabled admission control: In this experiment, we choose two different system configurations in the application server: One is our system with enabled admission control and the other is with disabled admission control. Each client will send a sequence of periodic tasks. The task arrival from each client is a Poisson process. We vary the task arrival rate from 0.10 per sec to 0.40 per sec. Each task life time is exponentially distributed, and each task includes 4 invocations in a life time in average. The period for invocation arrival in a task is 1.250 sec and all invocations in a task have a deadline 1.250 sec requirement.

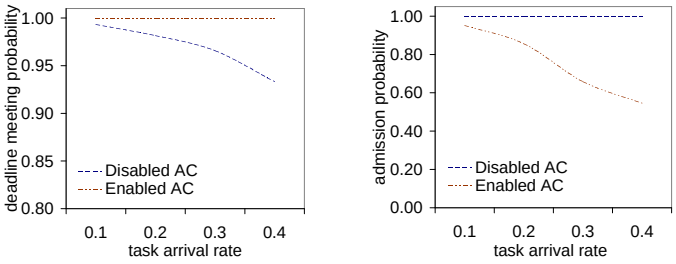

Fig. 5. Comparison of deadline meeting probabilities and admission probabilities

Fig. 5 shows deadline meeting probabilities for all invocations and admission probabilities for all task admission requests from a client (we run two clients and here we only report one and the other shows the similar phenomenon). As expected, as the task arrival rate increases, (i) the deadline meeting probability is always 100% and the admission probability decreases in the system with enabled admission control; (ii) the deadline meeting probability decreases and admission probability is always 100% in the system with disabled admission control. The data show that the admission control mechanism really makes any invocation in all admitted tasks meet its end-to-end deadline requirement by rejecting some task admission request.

b) Our system vs. a system without using component technology: In this experiment, we measure the overhead introduced by component technology in real-time systems. We evaluate the delay performance for a periodic task in: (i) a pure real-time RMI system; (ii) our designed real-time component-based system (admission control is disabled). We consider one of real-time components used in previous experiment and also choose the same implementation of its method in the pure real-time RMI system. The period of invocation arrival for this task is 0.600 sec. A client will invoke the periodic task either to the pure real-time RMI system or to our real-time component-based system.

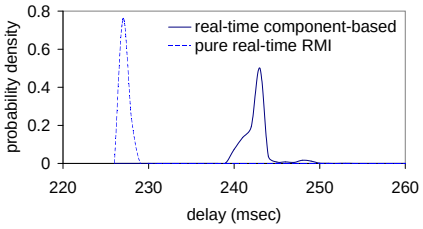

Fig. 6. Comparison of delay probability distribution function

Fig. 6 shows the sampled distribution function of method invocation delays. The average delays are 0.227 sec and 0.243 sec for the pure real-time RMI system and our system, respectively. The data show that the average delay is 0.0154 sec larger in our system than in the pure real-time RMI system. Component technology introduces little overheads to method invocation delays.

VI. Future Work

In this work, we developed a real-time component-based system, where components are truly reusable in providing both functional and real-time services. Furthermore, we addressed how to build real-time applications and real-time components based on our proposed resource overlay framework. The future work lies in two directions: (i) How to specify good real-time component interfaces? Basically, given the application task arrival pattern, we could optimally specify real-time service interfaces; (ii) How to improve the resource utilization? We could extend deterministic real-time guarantees to statistical real-time guarantees to achieve this.

References

[1] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming, 2nd Edition*. New York: Addison-Wesley / ACM Press, 2002.
[2] A. Pasetti and W. Pree, "The component software challenge for real-time systems," in *Proceedings of the First International Workshop on Real-Time Mission-Critical Systems*, Scottsdale, AZ, Nov./Dec. 1999.
[3] Object Management Group, "Realtime CORBA joint revised submission," March 1999, oMG Document orbos/99-02-12 ed.
[4] D. C. Schmidt and F. Kuhns, "An overview of the real-time CORBA specification," *Computer*, vol. 33, no. 6, pp. 56–63, 2000.
[5] D. C. Schmidt *et al.*, "Real-time CORBA with TAO," http://-www.cs.wustl.edu/∼schmidt/TAO.html.
[6] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An aspect-based composition tool for real-time systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2003.
[7] J. Liu, *Real-Time Systems*. New Jersey: Prentice Hall, 2000.
[8] Sun Microsystems, "Enterprise JavaBeans technology," http://java.sun.com/products/ejb.
[9] JBoss, http://www.jboss.org.
[10] TimeSys, "TimeSys Linux 3.1," http://www.timesys.com.