# Component Customization for Object-Oriented Distributed Real-time Software Development*

Stephen S. Yau and Fariaz Karim
Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287, USA
Email: {yau, karim}@asu.edu

**Abstract**

*To apply the component-based approach to distributed real-time software development, it is necessary to identify components based on both functional and real-time requirements. Since a component may be acquired from external sources, it becomes necessary during integration to ensure that a component satisfies the real-time requirements of the target application software. Since these requirements vary, a component should be customizable so that during integration it can adapt itself to the specific real-time requirements of the target-distributed software. To facilitate such activities, it is preferable to have components that are capable of performing self-customization using a set of built-in services. In this paper, an object-oriented real-time component framework and two built-in customization services are presented to address the specified issues.*

**Keywords**: Component-based distributed software, integration, customization, real-time, object-oriented component framework, built-in services

## 1. Introduction

In component-based software development (CBSD) approach [1,2], components can be acquired from various sources with possible customization to fit specific requirements, and then be integrated to build the application software. Currently, the availability of standardized middleware and component models, such as CORBA, COM, etc. facilitate the development of object-oriented distributed software in a heterogeneous environment. However, current middleware has serious limitations with respect to object-oriented real-time distributed software development [1-4].

Integration in component-based software development is a difficult task due to many incompatibilities that arise during integration [5], especially in case of distributed real-time software development due to the issues related to different non-functional requirements (e.g., difference in the deadlines, event-handling, synchronization, and resource requirements, etc.) in addition to the functional requirements of the target application software. Before the actual integration is performed, it is important to guarantee that the constructed software will satisfy the specified real-time requirements. On the other hand, it is also desirable that a component is reusable in different distributed real-time software systems. To address both issues, a component should be customizable during integration to fit itself into specific real-time requirements. Based on this perspective, it is obvious that component customization is a necessary activity in the overall integration process, not an optional one, and that components need to have a set of specific built-in services to facilitate customization. In this paper, we will present a framework and associated built-in customization processes to facilitate the integration in object-oriented distributed real-time software development. Our approach to component customization during integration fits seamlessly with an integration process, such as the ones described in [1,6].

## 2. Our Approach

Since the customization process is a part of our overall component integration process, we need to discuss it briefly. The integration process is general in the sense that it can be applied to both real-time and non real-time distributed software development. Moreover, a principal goal of designing the integration process is to make it independent of any specific middleware implementation or system architecture. Our integration approach can be summarized in the following steps:

1) Specify the target architecture, including its real-time requirements, such as task frequency, priorities, and deadlines, etc.

---

*This work is supported in part under a collaborative research agreement between Arizona State University and Fujitsu, Ltd.

2) Partition the specification from 1) into smaller parts based on the nature of the collaborations among the components of the target architecture.
3) For each sub-specification in the partitioned specification from 2), choose a candidate component based on the desired functionality.
4) Perform the compatibility check of the candidate components based on the specification.
5) Based on the results from 4), customize the candidate components to make them compatible with the target architecture.
6) Integrate the customized components using *Distributed Connectors* (DC) [6] to generate the distributed real-time software.

The component customization process corresponds to Step 5) of our integration approach. In this paper, we will focus our discussion on two specific aspects of component customization: task-priority of the target architecture, and exception handling policy of the target architecture. The overall customization approach related to these two aspects can be outlined as follows:

5.1) The *Assigned System Tasks* ($AST_c$) based on the requirements of the target architecture is supplied to the candidate component. An $AST_c$ *t* of a component *c* means that *t* has been identified as a required task in the target architecture (to satisfy a specific requirement), and *c* has been chosen to provide the functionality for *t*.

5.2) The specification of the exceptional conditions and the corresponding exception handling policy of the target architecture are supplied to the component.

5.3) The $AST_c$s from Step 5.1) are mapped onto the internal tasks of the component. The internal tasks are used to implement the services published by the component.

5.4) The priority of the internal tasks of the candidate component is adjusted and their invocation sequences are configured based on the results from Step 5.3). If a conflict occurs, then a runtime-monitoring scheme is used to resolve the conflict.

5.5) Based on the information from Step 5.2), the internal events, which may potentially lead to exceptional conditions, are monitored inside the component. If such a condition arises, it is handled according to the specification from Step 5.2).

At the end of Steps 5.4) and 5.5) a candidate component is customized based on the task priority and the exceptional conditions of the target architecture respectively. In the remainder of the paper, these customization processes will be denoted as *CPS* and *CEH* respectively.

We will present in Section 3 an object-oriented real-time component framework for facilitating component customization. In Sections 4 and 5, we will describe the layered architecture of the framework. Both *CPS* and *CEH* customizations will be discussed in Sections 6 and 7. An example will be presented in Section 8 to illustrate the use of the component framework and the CPS customization in developing component-based distributed real-time network management software. We will discuss the implementation issues in Section 9.

## *3. Real-time Component Framework*

To facilitate customization, candidate components should have some specific built-in services that can be systematically used during component integration. These built-in services also need to be published by the components so that an integration tool [1,2] can easily identify if a candidate component has the capability to customize itself. To address these issues, the following desirable properties of a component has been identified:

a. Appropriate interface for specifying the latency and the required resources for each published service.
b. Support for easy identification of the outgoing events generated by the component. The interface must also publish the nature of the event generation (e.g. periodic or aperiodic).
c. Built-in services for performing schedulability analysis to identify self-compatibility based on the target architecture.
d. Necessary implementation to enforce the consistency of priorities of the internal tasks with that of the target architecture.
e. Suitable support for handling different types of exceptional conditions as specified by the requirements.

Among these properties, the first two are the most fundamental for component to be used in a distributed real-time software. The information presented in the interface is necessary for determining if a component can meet the minimum timing requirements. Properties c, d, and e essentially correspond to a component's ability to identify if it is suitable to be used in a specific distributed real-time software system. In particular, the requirements specified in properties d and e are important since the semantics of the target architecture should be

preserved inside the component to guarantee the real-time requirements of the target software.

Based on the above discussion, we will present an object-oriented real-time component framework to satisfy the specified properties. The key difference between our framework and the concept of a framework in general is discussed in [6].

A *Real-time Component Framework* (*RCF*) is a specialized *Distributed Component Framework* (*DCF*) [6]. An *RCF* serves as a reusable architecture of a distributed real-time component, and implements a set of common services for facilitating various activities related to component integration and customization. An *RCF* can be used as a reference to generate suitable components for distributed real-time software development.

Just as a *DCF*, an *RCF* separates the run-time operations (i.e. service invocations) of a component from its integration and maintenance-time operations (i.e. schedulability analysis, customization, extension of functionality, etc.) by distributing them over several interfaces. The framework uses a layered architecture to isolate the implementation of the common services from that of its run-time services, which usually are different depending on the functionality of the component. An *RCF* can be used to generate both *atomic* and *composite* [6] components to address hierarchical composition of complex components from simpler ones. It provides an *architecture-query* mechanism to automatically generate information about a component's internal structure to facilitate extension of functionality during runtime.

As shown in Figure 1, the *RTManagement Layer* and the *RTSolution Layer* constitute the layered architecture of the framework. They are described in Sections 4 and 5 respectively.

## 4. The *RTManagement Layer*

The *RTManagement Layer* is primarily responsible for the following operations:

- Implement the common interfaces of a real-time component (described later).
- Forward externally generated service invocations to the appropriate objects or *subcomponents* [6].
- Manage references and other design-time information about the objects or *subcomponents* and connectors that reside in the *RTSolution Layer*.
- Implement several compatibility checking algorithms to identify the suitability of the component for use in a specific target architecture.
- Implement the necessary customization algorithms and apply them during integration.

An important part of *RCF* is its interfaces, which form the basis of a successful integration process. They are briefly described below.

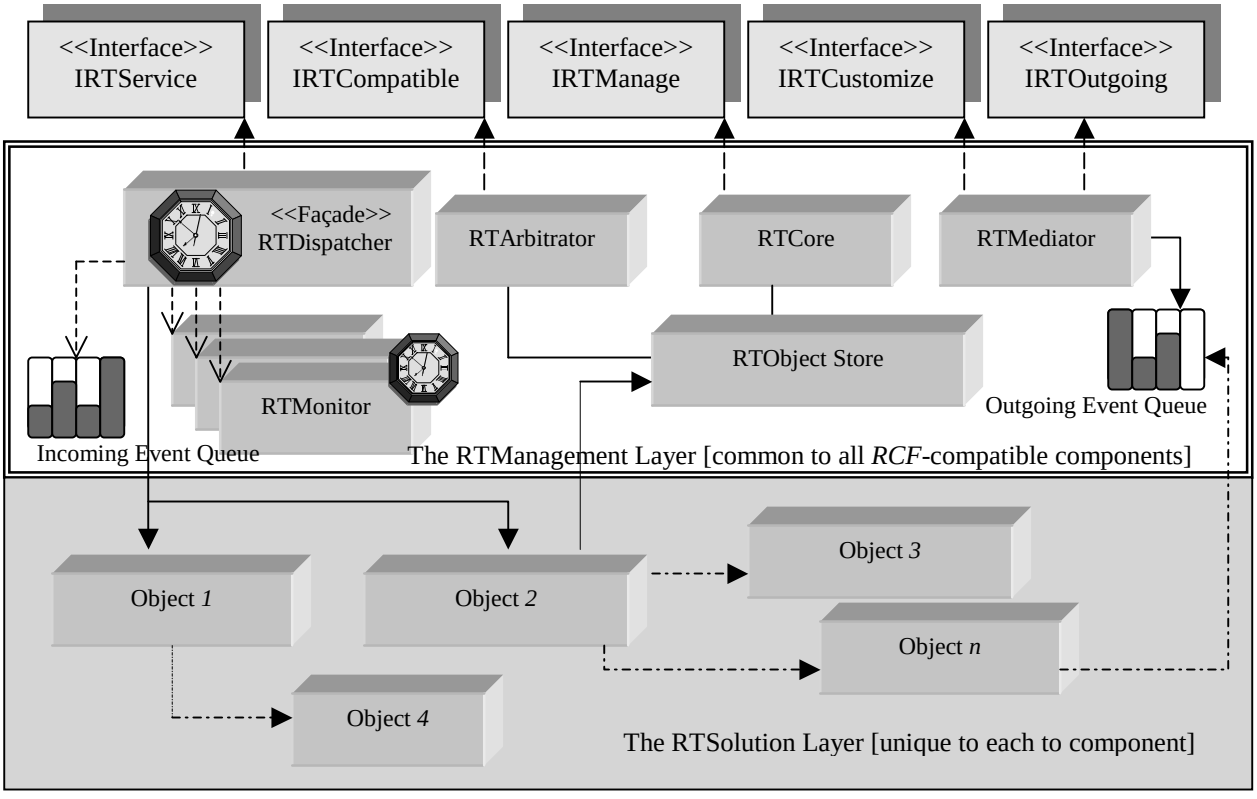


Figure 1: A Simplified View of the Real-time Component Framework (RCF)

a) ***IRTService*:** The *IRTService* interface publishes the services of the associated real-time component, which is generated by instantiating the framework. The services are declared in the following way:

$[ss_i, rt_i, [re^1_i, rc^1_i], [re^2_i, rc^2_i], \ldots [re^q_i, rcf^q_i]]$,
where
ssi = the method signature of the $i^{th}$ service,
$rt_i$ = the logical response time of the service,
$re^j_i$ = the $j^{th}$ resource required by method $ss_i$,
$rcf^k_i$ = the consumption function for the $j^{th}$ resource as required by method $ss_i$.

b) ***IRTCompatible*:** This interface takes the specification of the target real-time architecture [6]. The specification is derived in Step 2) of our integration approach, which is presented in Section 2. The interface provides separate methods for performing different types of compatibility checks.

c) ***IRTCustomize*:** The *IRTCustomize* interface is responsible for two services. It publishes one or more *plug-in* interfaces of the associated real-time component. The *plug-in* interfaces are used to publish services that are required by the component from its underlying environment or from external components. Moreover, the interface publishes two methods – *RTCustomizePriority* and *RTCustomizeException* to perform the *CPS* and *CEH* customizations as introduced in Section 2.

*RTCustomizePriority:* This method is used to perform the *CPS* customization. Among other input parameters, it requires the following data during its invocation:

a) A set of $AST_c$, $S_T$: $[t_1, t_2, \ldots t_n]$, where $S_T \subseteq S^*$, and S* is the set of all tasks identified in the system architecture.
b) A set of tasks with their priority and their deadlines, $S_{PT}$: $[[p_1, d_1], [p_2, d_2], [p_n, d_n] \ldots [p_n, d_n]]$, where $p_i$ is the priority and $d_i$ is the deadline of task $t_i$, and $t_i \in S_T$.
c) A set of task frequency, $S_{PD}$: $[[fp_1, fr_1], \ldots [fp_n, fr_n]]$, where $fp_i$ is the flag indicating if task $t_i$ is periodic or aperiodic, and $fr_i$ is the frequency of task $t_i$.
d) A set of task mapping relations $S_{mr}$: $[[t_1, ss_i], [t_2, ss_{i+1}], \ldots [t_n, ss_m]]$, where $t_i \in S_T$, and $ss_i$ is the $i^{th}$ service as published in the *IRTService* interface. The relations are used to identify the correspondence between an $AST_c$ with the method $ss_i$, which is published in the *IRTService* interface of the corresponding component.

*RTCustomizeException:* This method is used to perform the *CEH* customization. It mainly takes the following information as its input parameters –

$[edecl_i, pf_i, pr_i, ac_i, reh_j]$
where
$edecl_i$ = declarative specification of the conditions that will lead to exception i,
$pf_i$ = propagation flag for exception i, indicates if the exception should be self-contained or propagated,
$pr_i$ = the desired priority of the exception handler for exception i,
$ac_i$ = specification of the action that need to be taken if the exception is self-contained,
$reh_j$ = reference to a system-specific exception handler that need to be invoked when the exception occurs.

- ***IRTManage*:** It is mainly responsible for providing the *Architectural Reflection (AR)* service, which provides hierarchical query facilities to extend the functionality of a *composite* component in a distributed environment. Conceptually, the *AR* service can be thought of as a mechanism to systematically *opening* up a *composite* component in order to *peek* its inside. Any reconfiguration can be accomplished (after *peeking into it*) through the *Architectural Modification (AM)* service. These mechanisms can be used to remotely extend the functionality of a real-time component without shutting it down.

d) ***IRTOutgoing*:** In addition to specifying the services that a real-time component requires from its underlying environment, it publishes the internally generated outgoing events in the following way:

$[e_i, f_i, ph_i, exp_i]$,
where $e_i$ = the id of the $i^{th}$ event,
$f_i$ = the frequency of $e_i$,
$ph_i$ = period of $e_i$, if applicable,
$exp_i$ = a flag indicating if $e_i$ needs to be caught externally.

The data published in this interface is used mainly in two ways – to facilitate schedulability analysis during component integration, and to connect the candidate component with another component, a connector, or an adapter [1,2] that is interested in the published events.

As shown in Figure 1, the objects of the *RTManagement Layer* implement the described interfaces. The *RTDispatcher* object is actually a Façade [7], which connects the *IRTService* interface and the objects that implement the services of the component. It also invokes a set of *RTMonitor* objects, which are used to monitor the tasks in the

*RTSolution Layer*. The *RTArbitrator* implements the *IRTCompatible* interface and the compatibility checking algorithms. *RTCore*, on the other hand, provides both *AR* and *AM* services. The *RTMediator* object implements different customization algorithms, and mediate the communication between external components and the objects that reside in the *RTSolution Layer*. Finally, the *RTObject Store* serves as a repository of different data (e.g. deadlines, resource usage) concerning the objects of the *RTSolution Layer*.

## 5. The *RTSolution Layer*

This *RTSolution Layer* consists of objects or *subcomponents* that actually implement the services as published in the *IRTService* interface. If the component is *atomic*, then this layer consists of only objects. On the other hand, if the component is *composite*, then two or more *subcomponents* reside in this layer. The communication among the *subcomponents* is accomplished through one or more *connectors* using the *role-embedding* procedure. More information about *subcomponents*, *connectors*, and the *role-embedding* procedure can be found in [6].

The interactions between the *RTManagement* and the *RTSolution* layers follow three rules:

(a) All incoming service invocations must go through the path: *IRTService -> RTDispatcher -> $S_{oc}$*, where $S_{oc}$ is the object or *subcomponent* primarily responsible for providing the corresponding service.

(b) All events (including exceptions) generated in the *RTSolution Layer* must be either handled in the *RTSolution Layer* or be propagated outside the component through the *RTMediator* object.

(c) The *RTArbitrator*, the *RTDispatcher*, and the *RTCore* must communicate with the objects of the *RTSolution Layer* in a mutually exclusive fashion.

## 6. *CPS* Customization of Candidate Components

As described in Section 4, the *RTCustomizePriority* method of the *IRTCustomize* interface takes a set of input parameters to customize the priority of the internal tasks. The activity essentially corresponds to the adjustments of the priorities of the objects by considering inter-object invocation and synchronization. We first outline the actual procedure, and then briefly elaborate some of the non-obvious steps in a slightly detailed manner. The steps are initiated during component integration when the *RTCustomizePriority* method is invoked:

1) Generate the *invocation-graphs (IG)* of the objects or *subcomponents*, which reside in the *RTSolution Layer*.
2) Bind each *IG* with an $AST_c$.
3) Label each *IG* with the priority of the $AST_c$.
4) Identify any potential *priority-conflict* (*PoC*).
5) For each *PoC* identified at 4), resolve the problem by appropriately configuring the *RTMonitor* objects.

**Generation of *IG*:** Each *IG* captures the invocation sequence and synchronization constraints of the interaction among the objects in the *RTSolution Layer* as a result of an incoming method invocation. It can be generated by constructing the transitive closure for each service invocation inside the component. Each closure is saved into a separate *IG*. *IGs* need to be generated anew during component integration, since during integration a candidate component may connect to different external components and the topologies of the *IGs* may vary depending on the *IGs* of the external components.

**Binding and Labeling of *IG*:** In these phases, each *IG* is associated with one or more system tasks that are mapped to the services as specified by the mapping relation. For each mapping, the entire *IG* is then labeled with the corresponding priority of the system task. If multiple system tasks map to the same *IG*, multiple labeling is performed. The priorities of the labeled *IGs* then become the priorities of the corresponding object inside the *RTSolution Layer*.

**Detection of Priority Conflicts:** We consider a component *conflict-safe* if it is not guaranteed to violate the priority semantics of the system tasks. The necessary condition for this to be true is that each *IG* is mutually disjoint with each other. The necessary condition is extended for *IG*s when multiple system tasks are assigned to the same *IG*. The extra condition states that each labeled priority of an *IG* must be equal to each other.

Quite clearly, if the necessary condition is not met, there is a possibility that priorities of two system tasks will not be honored inside the component during runtime. The situation is referred to as *priority-conflict* (*PoC*). Figure 2 shows three system tasks $t_1$, $t_2$, and $t_3$ with priorities $p_1$, $p_2$, and $p_3$ ($p_1 > p_2 > p_3$) and their mappings into the corresponding *IG*s. Figure 2(a) shows a component $C_1$ in which a *PoC* exists since two *IG*s use the common object (painted black in the figure). Figure 2(b) illustrates another

case of *PoC* where in component $C_2$ the *IG*s are mutually disjoint, but multiple system tasks with different priorities are assigned to the same *IG* (i.e. the same method of the *IRTService* interface). Finally, Figure 2(c) shows a component that is *conflict-safe* for the particular target architecture.

The presence of a *PoC* may lead to a problem similar to priority inversion [8]. One possible approach is to detect such a problem early and label one of the conflicting *IG*s with higher priorities. This solution violates the semantics of the task priorities of the target architecture and may cause inefficiency if the corresponding system task has quite a large period.

***PoC* Conflict Resolution**: To resolve the conflict, an *RTMonitor* is assigned by the *RTDispatcher* to perform the following runtime actions:

a. In case of aperiodic tasks, when a conflicting task needs to be performed, the state of the conflicting object is saved before initiating the task inside the component. If another conflicting task with the higher priority arrives, then a new instance of the conflicting object is used. The new instance will be initialized with the saved state.

b. If the system tasks are periodic, then at the beginning of each period, a new instance of the conflicting object is activated using an *RTMonitor* object.

After the scheme is followed, the candidate component becomes customized for a particular distributed real-time architecture. If the component is reused in a different architecture, then the same procedure needs to be applied to adjust the internal priorities of the component.

## 7. *CEH* Customization of Candidate Components

The current component specifications, such as CORBA, allow specification of exception associated with an object interface. When such an exception is raised, it must be caught by either an external exception handler or the underlying environment. In case of component-based software, it should be noted that the system architecture itself might have some exceptional conditions, which should be detected and handled in a prioritized manner at the component level. Since a component may be reused in different real-time applications, the *definition of exceptional conditions* and the corresponding handling policies will vary. Accordingly, a real-time component must customize itself to preserve the semantics the exception handling of the target system architecture.

The *RTCustomizeException* method takes declarative specifications of the exceptional conditions of the target architecture. The actual customization is achieved using the following procedure:

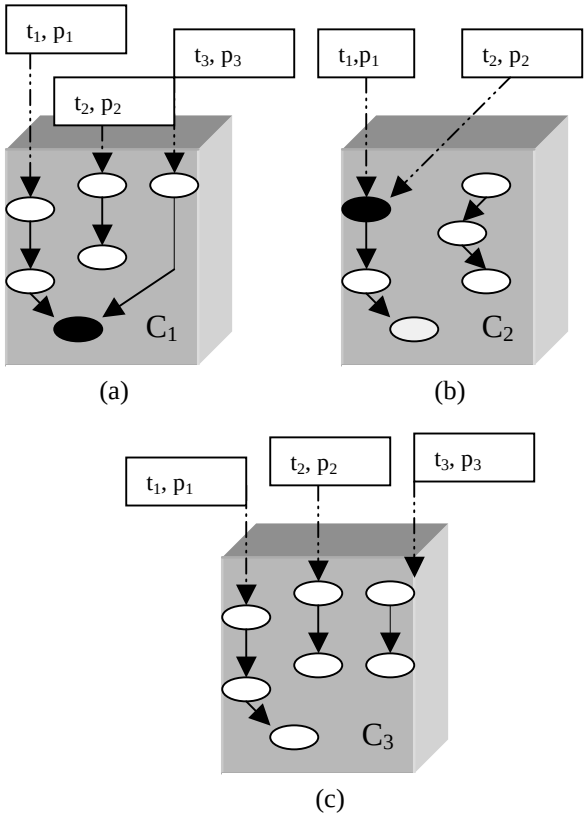


Figure 2: (a) A component with *priority-conflict* due to intersecting *IG*s, (b) A component with *priority-conflict* due to common mapping, (c) A component with no *priority-conflict*

a) Identify the exceptional conditions that involve the $AST_c$ of the candidate component. Even though there may be several exceptions in the architecture, a component only needs to concern itself with the ones that depend on its services.
b) Use deduction algorithms [9] and the *IG*s to identify the root conditions that will potentially lead to the exceptions, as identified in Step a).
c) Assign an *RTMonitor* object to monitor and detect such conditions.
d) Invoke either a system-provided handler or an internal handler (if available) to catch the exceptions.
e) If an internal handler is used, adjust its priority as specified in the *RTCustomizeException* method.

This scheme customizes the exception detection and handling policy of a component based on the target architecture. Installation of a system-provided handler is done through the *IRTCustomize* interface.

If the monitor identifies an exceptional condition, then the external handler is invoked. In this case, the underlying environment must guarantee that the external handler operates with the desirable priority.

## 8. An Example

In this section, we illustrate the use of real-time component framework and the *CPS* customization through an example. The example is adapted from [6], in which we illustrated the development of a distributed network management software system (*DNM*) by integrating several distributed components, which are generated based on the framework. In this section, we extend the original example to include some real-time requirements, and show how a specific candidate component in *DNM* uses its built-in customization facilities to make itself compatible with the target architecture.

The main operation of a DNM is to monitor a set of *network elements (NE)*, such as routers and gateways. The *NE*s generate different events that must be acknowledged and handled properly to keep the network free of congestion, or connected all the time. The requirements include concurrent and prioritized processing of events, separation of event handlers from the status monitors, fast propagation of event-processing status, and others. The additional real-time requirements are:

a) Each *NE* generates three different aperiodic events, which must be handled in a prioritized fashion. Let $t_i$ be the system task that needs to respond to event $e_i$. Let $p_i$ be the specified priority of $t_i$. Thus, the set of all tasks, S* = [$t_1$, $t_2$, $t_3$]. Let $p_1$, $p_2$, and $p_3$ be the priority of tasks $t_1$, $t_2$, and $t_3$ respectively. Also, let $p_1$ = 3, $p_2$ = 2, $p_3$ = 1 (1 being the lowest priority).
b) Let $d_i$ be the deadline of $t_i$. Also let, $d_1$ = 5 sec, $d_2$ = 5 sec, $d_3$ = 3 sec.
c) Let $st_i$ and $ft_i$ be the start time and the finish time of task $t_i$. Moreover, the exceptional conditions are, [$ft_1 - st_1 > 1$ sec], [$ft_2 - st_2 > 0.5$ sec], [$ft_3 - st_3 > 1.5$ sec]. The system will provide handlers for each exception.

Based on the requirements, three different components are chosen: *Event Dispatcher (CED)*, *Event Processor (CEP)*, and *Monitor (CM)*. *CED*s are installed on the *NE*s. They monitor the associated devices, generate, and dispatch the specified events to the *CEP* component. The *CEP* is responsible for processing the events, and forwarding the status to the *CM* component. The *CM* is responsible for providing GUI interface to present different information, such as the number of pending events and percentage of deadlines missed.

In this case, the *CEP* component is the most important of all, since it will be responsible for implementing the system tasks $t_1$, $t_2$, and $t_3$. Figure 3 shows the *CEP* component, its internal objects, (which will be hidden in the actual case), and mapping of the tasks $t_1$, $t_2$, and $t_3$ to the *IRTService* interface. The component is also connected with three external exception handlers, which are activated when any deadline is missed. The *RTSolution Layer* of the *CEP* component consists of objects $O_1$, $O_2$, $O_3$, $O_4$, and $O_5$. $O_1$, $O_2$, and $O_3$ implement the tasks $t_1$, $t_2$, and $t_3$ respectively.

During component integration, when the *RTCustomizePriority* method is invoked, the following operations will be performed:

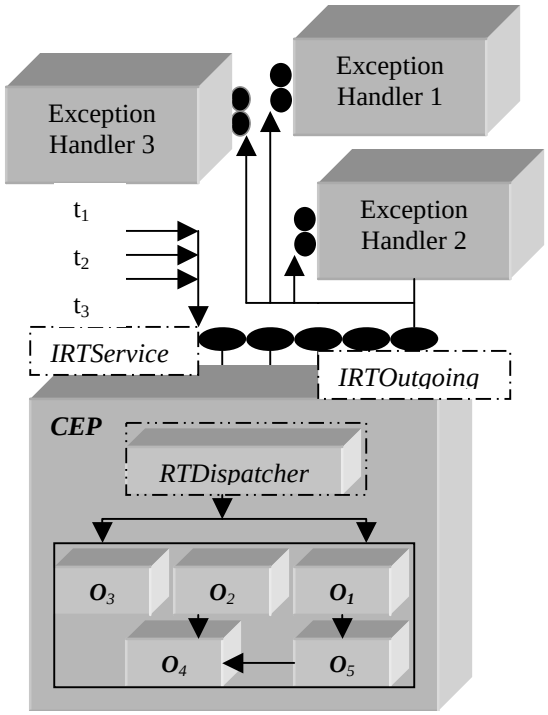


Figure 3: Component *CEP*

**--*IG* Generation:** $IG_1$, $IG_2$, and $IG_3$ will be generated, since only $O_1$, $O_2$, and $O_3$ directly implement the methods published in the *IRTService*.

**--Binding and Labeling of *IG*:** $IG_1$, $IG_2$, and $IG_3$ are bound to tasks $t_1$, $t_2$, and $t_3$. As a result, all the nodes will be labeled with the priority of the corresponding task. Figure 4 shows the priorities of the *invocation graphs*.

**--*PoC* Conflict Detection and Resolution:** A *priority-conflict* exists since both $O_1$ and $O_2$ transitively use $O_4$ to perform their respective

functionality. The conflict will become evident when $t_2$ is being serviced by $O_2$, and $t_1$ arises short after. Now $t_1$ must be handled by $O_1$, $O_5$, and $O_4$ with a priority = 3. However, since $t_2$ is being serviced, $O_4$ is executed with the priority = 2. To resolve this, two

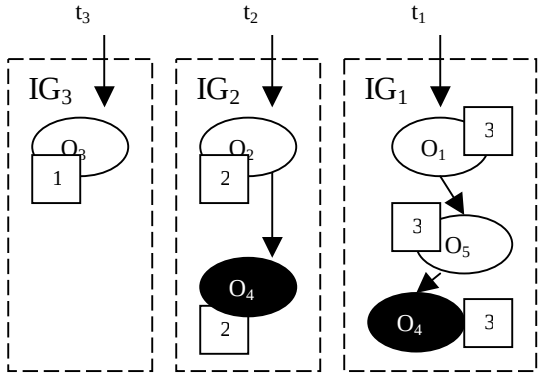


Figure 4: Labeled *IGs* of Component *CEP* with a priority conflict in $O_4$

instances of $O_4$ will be used. Whenever $t_2$ arrives, the state of $O_4$ will be saved. If $t_1$ arrives (which has a higher priority) while $t_2$ is being serviced, then the second instance will be started with the cached copy of the state.

## 9. Implementation

The design of *RCF* is independent of any specific middleware technology. However, we have used COM during implementation due to its availability. The incorporation of the runtime timers and the deduction algorithms inside the framework are currently being implemented. Support for communication is achieved through both DCOM and WinSock2 library. We use WinSock2 library to access the underlying Quality of Service (QoS) of a 650-mb/sec Fujitsu ATM network. We also plan to use VxWorks when a compatible COM runtime environment becomes available. A CORBA-compliant implementation is also possible although it is our plan to wait for a commercial implementation of Real-time CORBA.

## 10. Discussion

A real-time component framework is presented to facilitate customization during component integration. The framework uses a layered architecture to separate the common services of a component from the implementation of its published services. Two built-in schemes are presented that perform customization of a component based on the requirements of the target software.

Future work includes the capability of customizing fault-tolerance into our component framework. We plan to relate the customization algorithms with compatibility checking methods to make the component integration process more systematic and coherent. A successful integration often depends on the availability of a precise specification of the target architecture. Although several Architecture Description Languages (ADL) [10] exist today, they are not suitable to express the complexity and the semantics (e.g. exception handling policy) of a distributed real-time architecture. Thus, it is also our plan to derive a new ADL for specifying distributed real-time software systems.

***References*:**


[1] S. Yau and B. Xia, "An Approach to Distributed Component-based Real-time Application Software Development", *Proc. 1st IEEE Int'l Symp. Object-oriented Real-time Distributed Computing (ISORC 98)*, April 1998, pp. 275-283.

[2] S. Yau and B. Xia, "Object-Oriented Distributed Component Software Development Based on CORBA", *Proc. 22nd Int'l Computer Software and Applications Conf. (COMPSAC 98)*, August 1998, pp. 246-251.

[3] C. Gill, et al., "Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-time Systems", *Proc. 1st Int'l Workshop on Real-time Mission-Critical Systems: Grand Challenge Problems*, Phoenix, USA, November 1999.

[4] Object Management Group, "Real-time CORBA FTF", *OMG TC Work in Progress*, November 1999.

[5] D. Garlan, et al., "Architectural Mismatch: Why Reuse Is So Hard", *IEEE Software*, Vol. 12, No. 6, November 1995, pp. 17-26.

[6] S. Yau and F. Karim, "Integration of Object-Oriented Software Components for Distributed Application Software Development", *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, December 1999, pp. 111-116.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[8] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Trans. on Computers*, September 1990, pp. 1175-1185.

[9] J. Goubault-Larrecq, *Proof Theory and Automated Deduction*, Kluwer Publishers, 1997.

[10] N. Medvidovic and R. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", *Proc. 6th European Software Engineering Conference*, 1997, pp. 60-76.