

# Real-Time Scheduling Techniques for Implementation Synthesis from Component-Based Software Models

Zonghua Gu and Zhimin He

Dept. of Computer Science  
University of Virginia  
Charlottesville, VA 22903, USA  
{zg4v, zh5f}@cs.virginia.edu

**Abstract.** We consider a class of component-based software models with interaction style of buffered asynchronous message passing between components with ports, represented by UML-RT. After building a logical software model, it is necessary to synthesize a multi-threaded implementation that runs on a given target hardware platform and satisfies timing constraints. Commercial code generators generate functional code, but ignore concurrency and timing issues. In this paper, we compare alternative multi-threading strategies for implementation synthesis from this class of software models, and describe real-time scheduling analysis techniques that are useful during design space exploration for implementation synthesis. We use the elevator control application to illustrate our analysis techniques.

## 1 Introduction

We consider a class of component-based software models with interaction style of buffered asynchronous message passing between components with ports. This programming style is prevalent in development of event-driven real-time software. One representative example is UML-RT, a UML Profile for an architecture description language based on Real-Time Object-Oriented Modeling (ROOM) [1], supported by CASE Tools from IBM Rational [2]. Another example is the Quantum Framework [3], which advocates this programming style without the need for expensive CASE tools. It has a number of benefits from a software engineering perspective, such as modularity, encapsulation, decoupling of interactions, etc. This programming style is ideally combined with event-driven middleware like CORBA Event Service as the application's communication substrate. One real-world application example is the Avionics Mission Computing software [4].

After building a logical software model, it is necessary to synthesize a multi-threaded implementation that runs on a given target hardware platform and satisfies timing constraints. Commercial code generators typically generate functional code, but ignore concurrency and timing issues. It is up to the designer to

choose a multi-threading strategy to ensure satisfaction of system timing constraints. In this paper, we compare alternative multi-threading strategies for implementation synthesis from this class of software models, and describe real-time scheduling techniques that are useful during design space exploration for implementation synthesis.

We use UML-RT as a representative example in the following discussions, but note that the analysis techniques discussed in this paper have much wider applicability to the general class of component-based software models with interaction style of buffered asynchronous message passing between components with ports.

This paper is structured as follows: section 2 discusses different implementation alternatives. Section 3 discusses real-time scheduling analysis for one runtime model. Section 4 uses the elevator control application to illustrate our analysis technique. Section 5 discusses pros and cons of different multi-threading strategies; Section 6 discusses related work, and section 7 draws conclusions.

## 2 Multi-Threading Strategies for Component-Based Software Models

It is typical for the runtime model to follow the *Run-To-Completion* (RTC) semantics for each component, that is, once triggered by a message at its input port, the component must execute the triggered action to completion before processing the next message. RTC is useful for reducing the number of concurrency bugs when a component can take part in multiple end-to-end scenarios. Messages can be assigned priorities and queued in priority order instead of FIFO order. Each OS thread processes incoming messages for the components assigned to it in a *priority-based, non-preemptive* manner, consistent with the RTC semantics. However, there can be preemptions between different threads in a multi-threaded system, that is, a component executing in the context of a higher-priority thread can preempt another component executing in a lower-priority thread.

It is important to distinguish between the concepts of design-level concurrency and implementation-level concurrency [5]. At the design level, each component conceptually contains its own logical thread of execution, but each logical thread is not necessarily mapped into an OS thread at the implementation level. Although it is possible for each component to have its own OS thread, it may incur too much context-switching overhead if there are a large number of components. There are a number of possible multi-threading strategies for implementation synthesis, as discussed below.

Suppose we have a logical model as shown in Figure 1, consisting of three components  $O_1, O_2, O_3$  and two end-to-end scenarios  $t_1, t_2$ . Each scenario consists of multiple subtasks, which are triggered actions executed by the components. Scenario  $t_1$  is initially triggered by a periodic timeout message with period 10 ms that triggers an action  $t_{11}$  in component  $O_1$ , which in turn sends a message to component  $O_2$  and triggers action  $t_{12}$  in  $O_2$ . Finally,  $O_2$  sends a message to  $O_3$  and triggers action  $t_{13}$ . We can view this scenario as a logical end-to-end

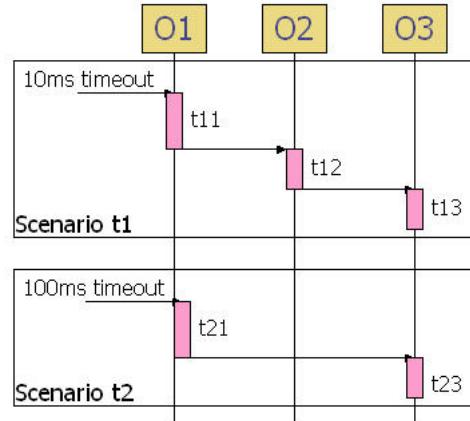
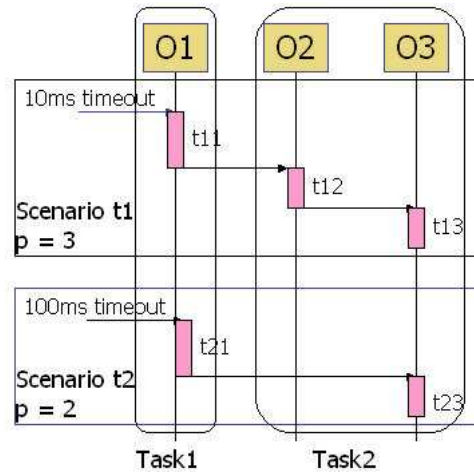


Fig. 1. An example application scenario.

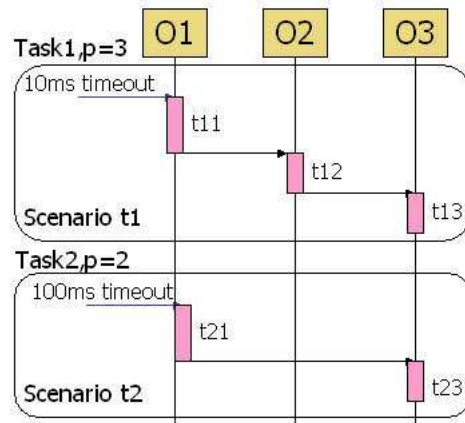
thread  $t_1$  consisting of three precedence-constrained subtasks  $t_{11}$ ,  $t_{12}$  and  $t_{13}$ . Similarly, scenario  $t_2$  is an end-to-end thread consisting of two subtasks  $t_{21}$  and  $t_{23}$ , triggered by a 100ms periodic timeout message. Given this logical model, how to implement it on a multi-threaded real-time operating system?

Despite the word *Real-Time* in its name, the designers of the component-based CASE tools have not put much emphasis on real-time issues when implementing a logical model on the target platform. The default runtime model is single-threaded, that is, all components are mapped into the same thread of execution. It is desirable to introduce more parallelism and concurrency into the system to improve predictability, by adopting a multi-threaded execution architecture. Commercial code generators, e.g., UML-RT code generator from IBM Rational, typically provide options for creating multiple threads, each containing one or more components. Each thread is assigned a fixed priority. Some authors have proposed alternative runtime models, as discussed below.

1. **Component-Based Multi-threading, Scenario-Based Priority-Assignment (CMSP)** This is proposed by Saksena in [5]. As shown in Figure 2, one or more components are grouped into the same thread. Priorities are associated with the end-to-end scenarios, and the thread priorities are adjusted dynamically to maintain a uniform priority across each application scenario.
2. **Scenario-Based Multi-Threading, Scenario-Based Priority-Assignment (SMSP)** This is proposed by Saehwa Kim in [6]. As shown in Figure 3, each application scenario is mapped into a separate thread with uniform priority.
3. **Component-Based Multi-threading, Component-Based Priority-Assignment (CMCP)** As shown in Figure 4, one or more components are grouped into a thread with uniform priority. The figure only shows one of many possibilities for grouping components into threads. Two extreme cases are mapping all components into a single thread, or mapping each component into its own thread.



**Fig. 2.** *Component-Based Multi-threading, Scenario-Based Priority-Assignment (CMSP).* Note that we use the words *thread* and *task* interchangeably in this paper.

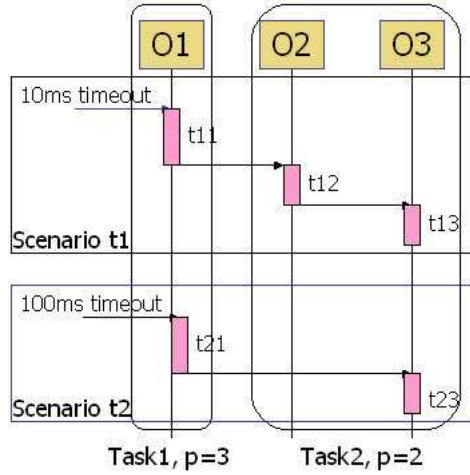


**Fig. 3.** *Component-Based Multi-Threading, Scenario-Based Priority-Assignment (SMSP).*

4. **Scenario-Based Multi-Threading, Component-Based Priority-Assignment (SMCP)** Even though this combination is conceptually possible, we do not know of any real applications that adopt it, so we will not consider it further.

*Rate Monotonic Analysis (RMA)* [7] is a well-known analysis technique for determining schedulability of a set of real-time tasks/threads, which must satisfy the following set of assumptions: each task must

1. be preemptively scheduled.
2. be independent.



**Fig. 4.** *Component-Based Multi-threading, Component-Based Priority Assignment (CMCP).*

3. be periodic.
4. have bounded *Worst-Case Execution Time* (WCET).
5. have uniform, static priority.

For scenario-based priority assignment, RMA assumptions are satisfied since each end-to-end scenario can be viewed as a task with uniform, static priority. But for component-based priority assignment, RMA assumptions are not satisfied, since each end-to-end task consists of multiple subtasks of varying priority. We describe real-time scheduling analysis techniques for this situation in the next section.

### 3 Real-Time Scheduling Techniques for CMCP

Consider a software model consisting of  $m$  components  $O_1, O_2, \dots, O_m$ , and  $n$  end-to-end scenarios, where each scenario is mapped into an *end-to-end virtual thread*, forming the taskset  $\tau_1, \tau_2, \dots, \tau_n$ . Here we use the word *virtual* to denote the fact that each end-to-end thread consists of multiple segments of subtasks distributed over different OS threads. Each end-to-end thread  $\tau_i, i = 1, \dots, n$  cuts through one or more components, and triggers an action within each component, forming a chain of subtasks  $\tau_{i1}, \dots, \tau_{im(i)}$ . We use  $O(\tau_{ij})$  to denote the component that the subtask  $\tau_{ij}$  belongs to, and  $PO(\tau_{ij})$  to denote the passive objects that  $\tau_{ij}$  accesses. Each subtask  $\tau_{ij}$  is actually an event-triggered action within a component  $O(\tau_{ij})$ . Each subtask  $\tau_{ij}$  is characterized by parameters  $(C_{ij}, P_{ij})$ , where  $C_{ij}$  is its worst-case execution time, and  $P_{ij}$  is its priority. Each end-to-end thread  $\tau_i$  has an end-to-end deadline  $D_i$ .

The task model is very similar to the task model of end-to-end threads with subtasks with varying priority, as described by Harbour, Klein, Lehoczky in [8].

We call the schedulability analysis algorithm introduced in [8] the *HKL algorithm*. However, in order to be applicable to CMCP, the HKL algorithm needs to be adapted to take into account blocking time caused by the multiple subtasks sharing common components and the Run-To-Completion (RTC) semantics. A component may be involved in multiple sub-tasks within one end-to-end thread, or in multiple end-to-end threads. Due to RTC, a subtask may suffer a blocking time equal to the largest execution time of other subtasks sharing the same component. Blocking time can also be caused by sharing of passive objects by multiple end-to-end threads. We do not model method invocations to passive objects as separate subtasks, since the passive object can be viewed as an extension of the invoking component, and inherits the thread and priority from it. But we do need to take into account blocking time caused by sharing of passive objects.

We first briefly describe the HKL algorithm. The *canonical form* of a task  $\tau_i$  is a new task  $\tau'_i$  with the same sequence of subtasks as  $\tau_i$ , but with strictly increasing priorities. One example transformation is a task-chain consisting of subtasks with priority sequence (8, 2, 5, 4, 3). The canonical form of this task-chain consists of priority sequence (2, 2, 3, 3, 3). It was proven in [8] that transforming a task into its canonical form does not affect its schedulability. This result allows one to check whether the canonical form of  $\tau_i$  is schedulable instead of  $\tau_i$  itself, which simplifies the analysis considerably.

Now define  $P_{min}(i)$  to be the minimum priority of all subtasks of  $\tau_i$ . The next step is to classify all tasks  $\tau_j, j \neq i$  according to their relative priority levels with respect to  $P_{min}(i)$ . For example, if the canonical form of  $\tau_i$  consists of a single segment of priority 18, and  $\tau_j$  consists of priority sequence (19, 10, 19, 10, 25, 10), or,  $(H, L, H, L, H, L)$ , where  $H$  stands for “higher”, and  $L$  stands for “lower”. There are five types of tasks [9]:

- Type 1, or  $H^+$ , tasks, with all subtask priorities higher or equal to  $\tau_i$ . These tasks can preempt task  $\tau_i$  multiple times.
- Type 2, or  $(H^+L^+)^+$ , tasks. The first subtask has higher priority than  $\tau_i$ , but it can only preempt  $\tau_i$  once, since it is followed by subtasks of lower priority. Multiple tasks of this type may preempt  $\tau_i$ , but only for the first segment. The non-first high-priority segments cause a *blocking* effect.
- Type 3, or  $((HL)^+H)$ , tasks. They differ from type 2 since they end with a high priority segment. We omit the discussion of type 3 tasks, since they do not appear in the example we consider here.
- Type 4, or  $(L^+H^+)^+L^+$ , tasks. The first subtask has lower priority than  $\tau_i$ . Any one of the following subtask segments can have a *blocking* effect on  $\tau_i$ , but only one such segment among all tasks of type 4 can have such a blocking effect.
- Type 5, or  $L^+$ , tasks. They have no effect on completion time of  $\tau_i$ , and can be ignored.

Suppose we are calculating response time of task  $t_i$ . To simplify discussions, let's assume the canonical form of  $t_i$  consists of subtasks of uniform priority  $P_i$ . Define  $H_1(i), H_2(i), H_4(i)$  to be the indices of all tasks of type 1, 2, 4, respectively.

For each  $j \in H_2(i)$ , let  $B_2(i, j)$  be the execution time of the *first*  $H^+$  segment of task  $\tau_j$ .  $B_2(i, j)$  denotes the *preemption* time caused by  $\tau_j$  to  $\tau_i$ . Then the total preemption time suffered by  $\tau_i$  is:

$$B_2(i) = \sum_{j \in H_2(i)} B_2(i, j)$$

For each  $j \in H_2(i) \cup H_4(i)$ , let  $B_4(i, j)$  be the *blocking* time suffered by  $\tau_i$ , caused by all  $H^+$  segments of task  $\tau_j$  of type 4, and all *non-first*  $H^+$  segments of task  $\tau_j$  of type 2. Then the total blocking time suffered by  $\tau_i$  is:

$$B_4(i) = \max(B_4(i, j) | j \in H_4(i) \cup H_2(i))$$

For a Type 2 task, only the first higher priority segment should be counted in  $B_2(i)$ , while the remaining segments should be counted in  $B_4(i)$ . Since multiple tasks of Type 2 can use their *first* segments to preempt  $t_i$ , therefore  $B_2(i)$  is a *sum* of them; while only one task of Type 2 or 3 can use its *non-first* segment to preempt  $t_i$ , therefore  $B_4(i)$  is a *max* of them.

In order to adapt the HKL algorithm to CMCP, we need to take into account additional blocking time  $B(i)$ :

$$B(i) = \max(C_{kl} | \forall k, l, j, k! = i, O(\tau_{kl}) = O(\tau_{ij})) + \max(C_{mn} | \forall m, n, j, m! = i, P_{mn} < P_{ij}, PO(\tau_{mn}) \cap PO(\tau_{ij}) \neq \phi)$$

where the first term denotes blocking time caused by other subtasks sharing the same component with some subtask of thread  $i$  due to the RTC semantics, and the second term denotes blocking time caused by other lower-priority subtasks accessing shared passive objects.

The equation for calculating the *Worst-Case Response Time* (WCRT) of task  $\tau_i$  is:

$$\text{WCRT}(i) = \text{WCET}(i) + B_2(i) + B_4(i) + B(i) + \sum_{j \in H_1(i)} \lceil \frac{\text{WCRT}(i)}{\text{Period}(j)} \rceil \cdot \text{WCET}(j) \quad (1)$$

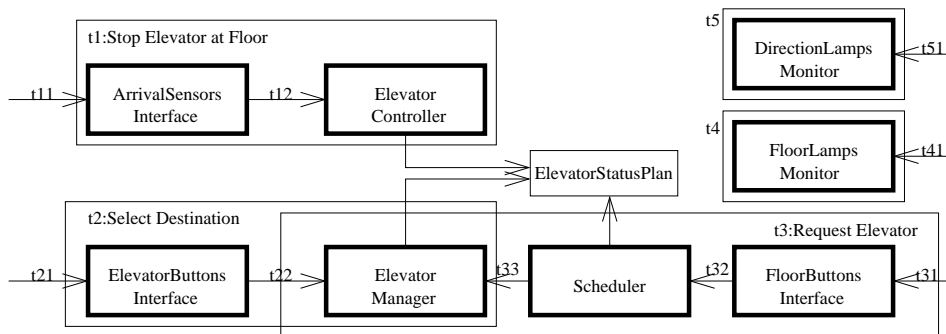
where  $\text{WCET}(i)$  is the worst-case execution time of  $\tau_i$ , and  $\text{Period}(j)$  is the execution period of  $\tau_j$  if it is a periodic task, or the minimum inter-arrival time of execution triggers for  $\tau_j$  if it is a sporadic task. The last term is preemption time caused by Type 1 tasks.  $\tau_i$  is schedulable if the calculated  $\text{WCRT}(i)$  is less than its deadline. This is a recursive equation that can be solved iteratively.

One limitation of our approach is that it can only handle linear task-chains, but not more general trees or graphs. It is an open research issue as to how to extend the HKL algorithm to deal with task-trees or graphs.

## 4 The Elevator Control Example

We use the elevator control system as an application example. Figure 5 shows the 8 components and 1 passive data object involved in a single-processor implementation. There are three end-to-end threads consisting of subtasks of varying priorities:

1. **Stop Elevator at Floor.** The elevator is equipped with arrival sensors that trigger an interrupt to the component *arrival sensors interface* when the elevator approaches a floor, which in turn sends a message *approaching floor* to the component *elevator controller*. The *elevator controller* invokes a synchronous method call on the passive data object *elevator status and plan* to determine whether the elevator should stop or not.
2. **Select Destination.** The user presses a button in the elevator to choose his/her destination, which triggers an interrupt to the component *elevator buttons interface*, which in turn sends a message *elevator request* to the component *elevator manager*. The *elevator manager* receives the message and records destination in the passive object *elevator status and plan*.
3. **Request Elevator.** The user presses the up or down button at a floor, which triggers an interrupt to the component *floor buttons interface*, which in turn sends a message *service request* to the component *scheduler*. The *scheduler* receives message and interrogates the passive object *elevator status and plan* to determine if an elevator is on its way to this floor. If not, the *scheduler* selects an elevator and sends a message *elevator request* to the component *elevator manager*. The *elevator manager* receives the message and records destination in the passive object *elevator status and plan*.



**Fig. 5.** The collaboration diagram for the single-processor elevator control system. Components are drawn with thick borders, and passive objects are drawn with thin borders.

Consider a building with 10 floors and 3 elevators. All end-to-end threads are interrupt driven, not periodic. In order to perform schedulability analysis, we estimate the worst-case arrival rate of the interrupts and use them as approximations for periods assigned to each task. For example, the **Request Elevator** scenario is assigned a period of 100 ms by assuming that all 18 floor buttons (up and down buttons for each floor, except the top and bottom floors) are pressed within 1.8 seconds, which is likely to be the worst-case arrival rate.

Task	Period	WCET	Priority	WCRT
<b><math>t_1</math>: Stop elevator at floor</b>				
$t_{11}$ : Arrival Sensors Interface	25	2	9	-
$t_{12}$ : Elevator Controller	25	5	6	19
<b><math>t_2</math>: Select Destination</b>				
$t_{21}$ : Elevator Buttons Interface	50	3	8	-
$t_{22}$ : Elevator Manager	50	6	5	38
<b><math>t_3</math>: Request Elevator</b>				
$t_{31}$ : Floor Buttons Interface	100	4	7	-
$t_{32}$ : Scheduler	100	12	4	-
$t_{33}$ : Elevator Manager	100	6	5	46
<b><math>t_4, t_5</math>: Other Tasks</b>				
$t_{41}$ : Floor Lamps Monitor	200	5	3	43
$t_{51}$ : Direction Lamps Monitor	200	5	2	48

**Table 1.** The taskset of the single-processor elevator control system. Higher number denotes higher priority.

We adopt CMCP (Component-based Multi-threading, Component-based Priority-assignment), where each component is assigned a fixed priority, and apply the schedulability analysis technique discussed in Section 3.

Table 1 shows the taskset of the elevator control system running on a single processor. The priorities are assigned in a rate monotonic fashion, that is, tasks with shorter periods are assigned a higher priority. In addition, the interrupt handler tasks [7], that is, the *Interface* subtasks, are assigned higher priorities than the other tasks in order to avoid losing any interrupts. Other priority assignment schemes are also possible. We do not address the priority assignment problem here, but only address scheduling analysis given existing priority assignments to subtasks.

As an example, let's consider the end-to-end task  $t_2$  **Select Destination**, which consists of two subtasks with execution time 3 and 6, priorities 8 and 5, respectively. Its canonical form is a single task with execution time 9 and priority 5. Other tasks can be classified as follows:

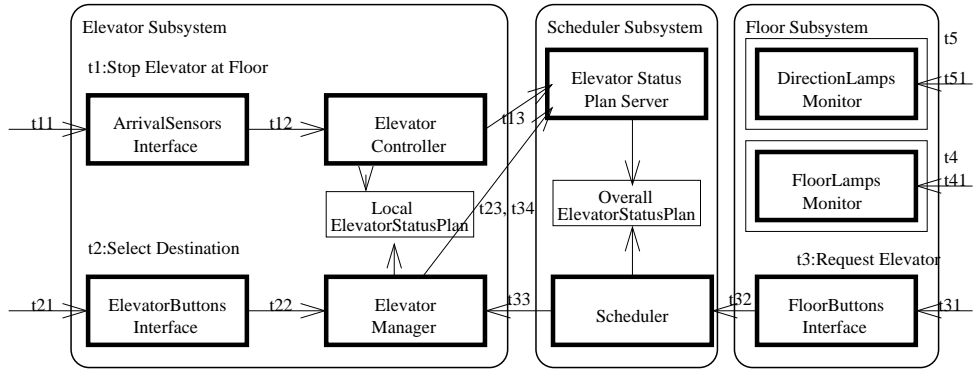
- $t_1$  is a type 1 task, with a single higher-priority segment with WCET 7.
- $t_3$  is a type 2 task, with a higher-priority segment  $t_{31}$  followed by lower-priority segments  $t_{32}$  and  $t_{33}$ .
- $t_4$  and  $t_5$  are type 5 tasks, with all segments having priorities lower than 5.

Blocking time  $B_2(2)$  caused by type 2 tasks is  $\text{WCET}(t_{31}) = 4$ . There are no Type 4 tasks. Blocking time due to RTC semantics is  $\text{WCET}(t_{33}) = 6$ ; blocking time due to shared passive objects is  $\max(\text{WCET}(t_{12}), \text{WCET}(t_{32})) = \max(5, 12)$ . We use Equation 1 to get:

$$\text{WCRT}(2) = \text{WCET}(2) + B_2(2) + B_4(2) + B(2) + \sum_{j \in H_1(2)} \left\lceil \frac{\text{WCRT}(2)}{\text{Period}(j)} \right\rceil \cdot \text{WCET}(j)$$

$$= 9 + 4 + 6 + \max(5, 12) + \lceil \frac{\text{WCRT}(2)}{50} \rceil \cdot 7 = 38$$

We can calculate WCRT for the end-to-end threads based on Equation 1, as shown in the WCRT column of Table 1. We associate the WCRT of the end-to-end thread with the last segment of the task in the table. No deadlines are missed, and the system is schedulable. Note  $t_4$  and  $t_5$  have relatively small WCRTs despite the fact that they have the lowest priority, since they do not suffer from blocking time caused by RTC semantics or shared passive objects.



**Fig. 6.** The collaboration diagram for the distributed elevator control system.

The single-processor system may become overloaded when more floors and elevators are involved. In order to be scalable to a large number of floors and elevators, the system needs to be redesigned to take advantage of multiple processors connected via a network, for example, the Controller Area Network (CAN). Figure 6 shows the system architecture. There is one *ElevatorCPU* for each elevator, and one *FloorCPU* for each floor. There is only one *SchedulerCPU* that is a central decision point for scheduling elevator requests, consisting of the component *scheduler* as well as another component *elevator status and plan server* for handling updates and queries from the components from the *ElevatorCPU* and *FloorCPU*. Each scenario spans multiple processors, and we need to take into account delays caused by scheduling of network packets. For a multi-processor elevator control system consisting of 12 elevators and 40 floors, we can calculate the end-to-end WCRT of distributed tasks by using Equation 1 as a subroutine for calculation of local WCRT on a single processor. We omit the details of this calculation due to space limitations, but the analysis results show that all tasks meet their deadlines.

From the above analysis, we conclude that the CMCP approach is adequate in terms of meeting system timing constraints. In this case, all three approaches result in meeting system deadlines, but it may not always be true. In general, we need to perform design space exploration, including choice of multi-threading

strategies and priority assignment, in order to determine the optimal implementation approach.

## 5 Experimental Evaluation

System Slack	Elevator Ctrl	Factory Automation	AMC ModalSP
Scenario-Based	0.16	0.21	0.21
Component-Based	0.24	0.38	0.05

**Table 2.** System slack of several application examples. System slack is defined as the maximum scaling factor for WCET of all threads to render the system unschedulable.

We have applied CMCP and SMSP to several application examples to analyze their schedulability under different multi-threading strategies. (From a real-time scheduling perspective, SMSP and CMSP both fit the assumptions of RMA, so we use SMSP as the representative case and compare it with CMCP.) We have assigned artificially-high WCETs for threads in order to push the systems to their limits of schedulability and highlight the differences between the two multi-threading strategies. As shown in Table 2, we calculated the system slack for three application examples. A larger system slack means that the system has more space to grow. For example, if we scale up the WCETs for all threads by 17%, it would render the Elevator Control application unschedulable with scenario-based multi-threading, but it would still be schedulable if we adopt component-based multi-threading. We can see that for Elevator Control and Factory Automation, component-based multi-threading leads to a larger system slack, while for the Avionics Mission Computing ModalSP (Modal Single Processor) scenario, scenario-based multi-threading leads to a much larger system slack.

From the evaluation results and observation of application characteristics, we can propose the following heuristics for helping the user choose a suitable multi-threading strategy: *If there is very little interaction between different application scenarios, then Scenario-Based Multi-Threading is appropriate. This is the case for Avionics Mission Computing scenarios, for example. However, if there is intensive interaction among different scenarios, then Component-Based Multi-Threading is more appropriate in order to avoid excessive locking and unlocking of shared components. This is the case for elevator control and factory automation examples.* For the elevator control example, the shared object *ElevatorStatusPlan* is the main cause of close interaction among scenarios *t1*, *t2* and *t3*.

Besides schedulability, there are also pros and cons to each approach from a software engineering perspective. CMSP requires the programmer to stick to a programming discipline of dynamically adjusting component priorities to reflect the priority of the currently executing end-to-end scenario. This approach hurts

the encapsulation of components by mixing system-level concerns (scenarios) with component-level concerns (components). It also involves runtime system-call overheads that may or may not be acceptable to certain resource-constrained embedded systems. Certain small RTOSes may not even provide APIs to dynamically change thread priorities. SMSP eliminates the need for dynamic priority adjustments, but creates shared data and necessitates error-prone concurrency control mechanisms, such as mutexes, semaphores and monitors. This breaks a key advantage of CMCP, which is to use buffered asynchronous message passing as the main communication mechanism among components instead of shared data in order to minimize the need for concurrency control. Note that even in CMCP, there are passive objects that are used to encapsulate shared data in addition to the components. The number of such passive objects should be minimized relative to the number of components.

Compared to CMSP or SMSP, CMCP has a number of advantages from a software engineering perspective, such as modularity, encapsulation, decoupling of interactions, mature tool support, etc. It is also the default runtime model implemented in UML tools, so a lot of legacy applications follow this model, and are not likely to be changed.

## 6 Related Work

Besides the work of Saksena [5] and Kim [6] discussed in detail in Section 2, there has been a lot of work on real-time analysis of component-based embedded software in the CBSE community. Muskens [10] presented a method for predicting runtime resource consumptions in multi-task component-based systems by expressing resource consumption characteristics per component, and combining them to do predictions over compositions of components based on end-to-end scenarios. Their work is targeted towards the Robocop component model. Sandstrom [11] introduced Autocomp, a component technology for safety critical embedded real-time systems, and discussed techniques for component-to-task mapping and task attribute assignment. Wall [12] proposed a method for impact analysis of adding new components to an existing product line based on Prediction-Enabled Component Technology. They considered two properties: end-to-end temporal property and version consistency property. Eskenazi [13] proposed a stepwise approach to predicting the performance of component compositions. Diaz [14] presented a predictable component model and a set of real-time analysis techniques based on mapping from the component model to SDL. Our work is unique in addressing the issue of one component participating in multiple end-to-end scenarios, which makes it desirable to adopt concurrency control methods such as Run-To-Completion, and the runtime model of Component-Based Multi-threading, Component-Based Priority-Assignment (CMCP).

There have been a lot of work in the real-time community on model-based and component-based design tools, conducted concurrently with and in relative isolation from the work of the CBSE community. In the past several years, DARPA has sponsored projects such as *Model-Based Integration of Embedded*

*Software* (MoBIES) and *Program Composition for Embedded Software* (PCES). Representative tools developed include CoSMIC [15], Virginia Embedded Systems Toolkit (VEST) [16], Time Weaver [17], Cadena [18], and the ESML-based Tool-Chain [19][20][21]. CoSMIC [15] uses the *Platform-Independent Modeling Language* (PICML) to enable developers to define component interfaces, QoS parameters and software building rules, and generate descriptor files that facilitate system deployment. PICML is designed to help bridge the gap between design-time tools and the actual deployed component implementations. VEST [16] is an integrated environment for constructing and analyzing component based embedded systems. *Aspect-checks* are used to check for cross-cutting non-functional properties, and *prescriptive aspects* are used to apply cross-cutting advice to design models. Time Weaver [17] is a software-through-models framework that decomposes inter-component relationships with an abstraction named *coupler*. Cadena [18] is an integrated development, analysis, and verification environment for *CORBA Component Model* (CCM) systems. The ESML-based Tool-Chain [20] provides an open and integrated development environment based on precise meta-modeling and the *Open Tool Integration Framework* (OTIF) [22] for easy plugin and semantic inter-operability of third party tools. All of the above work focus on static offline design and analysis. In contrast, Sharma [23] developed component-based dynamic QoS adaptations in distributed real-time and embedded systems by implementing QoS behavior as components that can be assembled with other application components. These projects mainly target the Avionics Mission Computing software [4] from Boeing, which follows the Scenario-Based Multithreading, Scenario-Based Priority assignment (SMSP) approach.

In embedded software development, UML models typically serve in an informal documentation role that the engineer refers to while writing code manually. This is one of the major motivations for developing domain-specific modeling languages and tools to replace UML in both the CBSE and real-time communities, in order to have a more formal, automated and integrated software development process. However, there has been recent progress on making UML more formal and suitable for modeling real-time embedded and component-based software. Some examples include UML-RT, UML 2.0 [24], UML Profile for Scheduability, Performance and Time [25] and the UML Profile for CORBA Component Model [26]. In particular, UML 2.0 has adopted the UML-RT concept of *components* communicating with message passing through *ports*. An interesting question to ask is, can we not give up on UML and develop custom, proprietary modeling notations, but leverage the body of work from the UML community to develop tools based on standards? One argument for preferring a custom modeling approach to UML is that we can achieve better domain-specificity by customizing the meta-model, which is more powerful and flexible than the UML profiling mechanism. Another argument is that embedded systems are so diverse that it is next to impossible to have one standard notation that is suitable for all application domains. For example, even though UML-RT is intended to be a general-purpose design tool, it has been mostly used for developing embedded

software in the telecommunications domain, which fits well with the interaction style of asynchronous message passing. We plan to investigate these interesting issues in our future work.

## 7 Conclusions and Future Work

In this paper, we have considered a class of component-based software models with interaction style of buffered asynchronous message passing between components with ports, which is a prevalent interaction style for large-scale complex real-time embedded software. Commercial code generators typically generate functional code, but do not take into account timing and scheduling issues. The runtime model of CMCP (Component-based Multi-threading and Component-based Priority-assignment) does not fit the assumptions of classic real-time scheduling theory, i.e., Rate Monotonic Analysis (RMA). Some authors have proposed alternative runtime models that can be analyzed with RMA. We take the alternative approach of adapting real-time scheduling theory to fit the runtime model of CMCP, instead of adapting the runtime model to fit real-time scheduling theory. This should make our approach more acceptable to industry than previous work in the literature.

We believe our work helps bridge the gap between a logical software model and its final implementation on the target platform, by giving the engineer real-time scheduling analysis techniques for evaluating different alternatives of generating a multi-threaded implementation from a logical software model. It focuses on the nonfunctional/real-time aspect of implementation synthesis, and is complementary to the existing code generators, e.g., UML-RT code generator from IBM Rational, which focus on the functional aspect of implementation synthesis. It is our future work to integrate our analysis techniques with commercial code generators. Even though the discussions in this paper are mainly based on UML-RT, our work has much wider applicability to the general class of component-based software models with interaction style of buffered asynchronous message passing between components with ports.

We have considered the problem of schedulability analysis given a system configuration of component-to-thread grouping and thread priority assignment, but it is still an open issue as to how to arrive at such a configuration. We did not deal with the design space exploration issues of how to group components into threads or assign priorities to threads. For the CMCP approach, the number of threads needs to be carefully managed. If there are too many threads, the context-switching overheads may be excessive; if there are too few threads, the blocking time may be too much due to insufficient parallelism. Priority assignment is also an important issue that needs to be considered for meeting system timing constraints. Exhaustive search is not feasible in general because the size of design space grows exponentially with the number of components or priorities. One possible future work is to apply optimization techniques such as branch-and-bound, simulated annealing and genetic algorithms for design space exploration,

in order to optimize design objectives such as minimizing the number of threads or minimizing response time for critical application scenarios.

## References

1. B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object Oriented Modeling*. Addison Wesley, 1994.
2. (2004) The IBM Rational website. [Online]. Available: <http://www-306.ibm.com/software/rational/>
3. (2004) The Quantum Framework website. [Online]. Available: <http://www.quantum-leaps.com/qf.htm>
4. Z. Gu, S. Kodase, S. Wang, and K. G. Shin, "A model-based approach to system-level dependency and real-time analysis of embedded software," in *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2003, pp. 78–85.
5. M. Saksena and P. Karvelas, "Designing for schedulability: integrating schedulability analysis with object-oriented design," in *Proc. IEEE Euro-Micro Conference on Real-Time Systems*, 2000, pp. 101–108.
6. J. Masse, S. Kim, and S. Hong, "Tool set implementation for scenario-based multi-threading of uml-rt models and experimental validation," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003, pp. 70–77.
7. M. H. Klein, T. Ralya, B. Pollak, and R. Obenza, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.
8. M. Harbour, M. H. Klein, and J. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. Software Eng.*, vol. 20, no. 2, pp. 13–28, 1994.
9. S. Tripakis, "Description and schedulability analysis of the software architecture of an automated vehicle control system," in *Proc. International Workshop on Embedded Software*, 2002, pp. 123–137.
10. J. Muskens and M. Chaudron, "Prediction of run-time resource consumption in multi-task component-based software systems," in *Proc. International Symposium on Component-Based Software Engineering, LNCS 3054*, 2004, pp. 162–177.
11. K. Sandstrom, J. Fredriksson, and M. Akerholm, "Introducing a component technology for safety critical embedded real-time systems," in *Proc. International Symposium on Component-Based Software Engineering, LNCS 3054*, 2004, pp. 194–208.
12. A. Wall, M. Larsson, and C. Norstrom, "Towards an impact analysis for component based real-time product line architectures," in *Proc. Euromicro Conference*, 2002, pp. 81–88.
13. E. Eskenazi, A. Fioukov, and D. Hammer, "Performance prediction for component compositions," in *Proc. International Symposium on Component-Based Software Engineering, LNCS 3054*, 2004, pp. 280–193.
14. M. Diaz, D. Garrido, L. M. Llopis, F. Rus, and J. M. Troya, "Integrating real-time analysis in a component model for embedded systems," in *Proc. EUROMICRO Conference*, 2004, pp. 14–21.
15. K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A platform-independent component modeling language for distributed real-time and embedded systems," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004.

16. 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 *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003, pp. 58–69.
17. D. de Niz and R. Rajkumar, "Time weaver: A software-through-models framework for embedded real-time systems," in *Proc. ACM Conference on Languages, Compilers and Tools For Embedded Systems*, 2003, pp. 133–143.
18. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *Proc. IEEE International Conference on Software Engineering*, 2003.
19. Z. Gu, S. Wang, S. Kodase, and K. G. Shin, "An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software," in *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2003, pp. 78–81.
20. —, "Multi-view modeling and analysis of embedded real-time software with meta-modeling and model-transformation," in *Proc. IEEE International Symposium on High Assurance Systems Engineering*, 2004, pp. 32–41.
21. Z. Gu and K. G. Shin, "Model-checking of component-based real-time embedded software based on corba event service," in *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2005.
22. G. Karsai, "Design tool integration: An exercise in semantic interoperability," in *Proc. IEEE Conference on Engineering of Computer Based Systems*, 2000.
23. P. K. Sharma, J. P. Loyall, G. T. Heineman, R. E. Schantz, R. Shapiro, and G. Duzan, "Component-based dynamic qos adaptations in distributed real-time and embedded systems," in *International Symposium on Distributed Objects and Applications*, 2004, pp. 25–29.
24. (2004) The Object Management Group website. [Online]. Available: <http://www.omg.org>
25. OMG, "Uml profile for schedulability, performance, and time specification," Object Management Group, Tech. Rep., 2003. [Online]. Available: <http://www.omg.org/technology/documents/formal/schedulability.htm>
26. —, "Uml profile for corba components specification," Object Management Group, Tech. Rep., 2004. [Online]. Available: <http://www.omg.org/cgi-bin/doc?ptc/2004-03-04>