Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Systems

M. Saksena*  P. Freedman†  P. Rodziewicz†

Abstract

In this paper we present our experiences in applying real-time scheduling theory to embedded control systems designed using ROOM (Real-time Object Oriented Modeling) methodology. ROOM has originated from the telecommunications community, and has been successfully applied to many commercial systems through the supporting case tool ObjecTime. It is particularly suitable for modeling reactive real-time behavior. Furthermore, it provides many other advantages through the use of object-orientation, and the use of executable models from which code may be generated quickly and efficiently. Since many real-time embedded control systems have significant reactive, event-driven, behavior, it is attractive to use ROOM methodology to develop such systems. However, the ROOM methodology does not provide tools to specify and analyze the temporal behavior as is required for the hard real-time components of embedded systems, and for which the real-time scheduling theory provides an analytical basis. In this paper, we show how real-time scheduling theory may be applied to ROOM models using a cruise control example to illustrate. The biggest challenge comes from minimizing the adverse effects of priority inversions. Our results are very encouraging, and we show that not only is it possible to apply real-time scheduling theory, but that it can be done very efficiently provided certain guidelines are followed in the design and implementation of the ROOM model.

1. Introduction

With the rapid advances in computing and communication technology, computerized control is being widely employed in many real-time control applications. Digital

*Department of Computer Science, Concordia University, Montreal, Canada, H3G 1M8. Phone: (514) 848-3024, Fax: (514) 848-2830, Email: manas@cs.concordia.ca.
†Centre de recherche informatique de Montreal (CRIM) Montreal, Quebec, Canada, H3A 2N4. Phone: (514) 840-1234, Fax: (514) 840-1244, Email: freedman@crim.ca.

1In a recent article entitled “The Challenges of Real-Time Software Design” [21], Selic and Ward use the words “event-driven” and “time-driven” to describe two basic ‘styles’ of real-time software. They note that “even though most real-time systems require a mix of the two styles, real-time software is usually constructed to suit the style required by the dominant application to the detriment of those in the other category” (p. 68).
esign time, about time-driven behavior of real-time systems, such as that exhibited by periodic computations associated with control laws. Nevertheless, as embedded control systems become more complex, the reactive behavior becomes harder to implement with ad-hoc techniques. Techniques based on finite state machines are naturally suited for modeling the reactive behavior of real-time systems, thus making ROOM/ObjecTime particularly attractive as a potential development methodology for real-time embedded systems. However, the integration of control structure associated with the reactive behavior and the dataflow structure associated with the time-driven behavior has not received much attention [17].

With this observation in mind, work began at CRIM in 1995-96 to investigate the suitability of ROOM/ObjecTime and its software development automation for the development of embedded control systems, in cooperation with Bombardier’s Transportation Equipment Group, one of North America’s leading mass transit companies. And in RTAS96, we described how ROOM models could be subjected to Generalized Rate Monotonic Analysis (GRMA) [15], using a simplified cruise control example [23]. The intent here was to promote GRMA to provide guidance, at design time, to the software development team. At the same time, a sister project began in 1996, jointly funded by Bombardier and the Canadian government to include the University of Sherbrooke and Concordia University. In this paper, we report on Concordia-CRIM collaborative work about guidelines for developing ROOM models and for their automated translations into implementations. Once again, we shall refer to a cruise control example but in a slightly modified form. In particular, we shall describe how key parts of software requirements associated with operator-induced changes to operating modes may be subjected to GRMA. As a result of attempting to applying GRMA to ROOM models, we have come up with a set of guidelines that can aid both tool developers in ensuring that their tools enable the application of real-time scheduling to system designs, as well as system designers in analyzing timing constraints and ensuring that their designs meet the timing requirements.

While our work is done in the context of a specific methodology (i.e., ROOM), we believe that the results and the observations are more generally applicable. In particular, the results developed here are equally applicable to Unified Modeling Language (UML) [7], an object-oriented modeling language being developed by using concepts from several object-oriented development methodologies (OMT, Booch, and OOSE), with statecharts at its heart.

The remainder of the paper is organized as follows. In Section 2 we present the key aspects of the ROOM modeling technique. In Section 3, we present the cruise control case study and develop a ROOM model for it. In Section 4, we develop some guidelines for design and implementation of ROOM models aimed to minimize priority inversions. In Section 5, we show how, based on the guidelines developed, a ROOM model design can be subjected to schedulability analysis. Finally, we present some concluding remarks in Section 6.

2. Overview of ROOM Concepts

The ROOM method adopts an operational approach to system analysis, design and implementation. It is based on establishing early operational models of the system and then refining them to implementation. It uses the concept of executable models which evolve from requirements to design to implementation. A ROOM executable model is a set of coherent structure and behavior views which can be compiled and executed on a variety of simulation and/or target platforms.

Modeling of systems with ROOM is performed by designing actors, which are encapsulated, concurrent objects, communicating via point-to-point links. Inter-actor communication is performed exclusively by sending and receiving messages via interface objects called ports. A message is a tuple consisting of a signal name, a message body (i.e., data associated with the message), and an associated message priority.

The behavior of an actor is represented by an extended state machine called a ROOMchart, based on the statechart formalism [11]. Each actor remains dormant until an event occurs, i.e., when a message is received by an actor. Incoming messages trigger transitions associated with the actor’s finite state machine. Actions may be associated with transitions, as well as entry and exit points of a state. The sending of messages to other actors is initiated by an action. The finite state machine behavior model imposes that only one transition at a time can be executed by each actor. As a consequence, a run-to-completion paradigm applies to state transitions. This implies that the processing of a message cannot be preempted by the arrival of new (higher priority) message for the same actor. However, as explained later, in a multi-threaded implementation, the processing may be preempted by other higher priority threads.

ROOM supports the notion of a composite state, which can be decomposed into substates. Decomposition of a state into substates can be taken up to an arbitrary level in a recursive manner. The current state of such a system is defined by a nested chain of states called a state context. The behavior is said to be simultaneously “in” all of these states. Transitions on the innermost current state take precedence over equivalent transitions in higher scopes. An event for which no transition is triggered at all levels of the state hierarchy is discarded, unless it is explicitly deferred.

ROOM also provides the concept of a layered architec-
A layer provides a set of services to the entities in the layer above. The linkage between layers is done at discrete contact points which are called service access points (SAPs) in the upper layer which uses the services, and service provisioning points (SPPs) at the layer providing the services. Each service access point is connected to a service provision point in the layer below (there can be a many to one mapping), and the end points of each such connection must have matching service points.

The bottom layer in ROOM models is provided by the ROOM virtual machine, which provides, among other things, a communications service and a timing service. The communications service provides the services to establish and manage connections between ROOM actors. The timing service may be used to set and cancel timers, both one-shot and periodic [19]. The ROOM virtual machine is also responsible for interfacing to other external (non-ROOM) environments such as specialized hardware or other software components and systems.

2.1. Run Time Systems and ObjecTime Toolset

ROOM run-time systems provide an implementation of the ROOM virtual machine, and are responsible for providing the mechanisms that support the ROOM paradigm as well as the services needed by ROOM models. The ObjecTime toolset (http://www.objectime.com) is a CASE tool that provides a fully integrated development environment to support the ROOM methodology, with features such as graphical and textual editing for actor construction and C++ code generation from the model [20]. The ObjecTime toolset includes a micro run-time system microRTS [1], which is linked with the application code to provide a standalone executable that may be run on either a workstation (emulation) environment, or on a target environment with an underlying real-time operating system such as VxWorks, QNX, pSOS, and VRTX.

In ROOM, actors are potentially concurrent objects, and also have a private address space. In implementing ROOM models, one must deal with the mapping of ROOM actors to the underlying operating system’s execution abstractions. Operating systems typically provide two such abstractions: (1) a heavy weight process, which has a private address space, and communicates with other processes using inter-process communication services such as sockets, and (2) a light weight thread, which shares memory with other threads within the same process. Since communication between processes is expensive, and ROOM models rely heavily on inter-actor communication, mapping actors to processes would be inefficient in most cases. In general, a ROOM model is better implemented as a (potentially) multi-threaded process, where actors are grouped into one or more lightweight threads. Each thread can then be implemented as a message handler, which executes a main loop in which it waits for messages arriving on its ports, and processes the arriving messages in priority order. This ensures that message processing cannot be preempted by another message arrival within the same thread consistent with the run-to-completion semantics of ROOM. However, a thread may be preempted by other threads depending on thread priorities and the scheduling of threads by the underlying operating system.

2.2. Transactions and Timing Constraints

ROOM provides the notion of scenarios to describe system behavior. A scenario may be pictorially depicted using message sequence charts [13], and additional annotations may be used to indicate timing constraints [4]. However, ROOM itself does not provide any mechanisms to specify and enforce timing constraints. As scenarios correspond to broad system descriptions, they are not particularly appropriate for specifying timing constraints. Instead we use the term transaction to describe end-to-end computations on which timing constraints such as periodicity and deadlines may be specified. A transaction is defined as a causally chained sequence of events triggered by an external incoming (stimulus) event, and possibly terminating with an external outgoing (response) event. The triggering events may be aperiodic (often corresponding to device interrupts) or periodic (generated by a timing service). We allow the following timing constraints to be specified on a transaction:

- An activation period, which represents either the inter-arrival time of the periodic timer, or a minimum inter-arrival time for aperiodically triggered transactions.
- An end-to-end deadline on the response time of the transaction.

An example of a transaction depicted using Message Sequence Charts is shown in Figure 1. All possible transactions in a given system can be mechanically generated by starting from the external incoming events, and then forming event sequences by recursively considering the output event set for the tasks handling the events. Of course, many such possible transactions may be meaningless since the appropriate triggering conditions may not hold. The definition of transactions allows two transactions to be triggered by the same external incoming event, and also have a common initial event sequence. Such transactions would then have the same activation period, but may have different deadlines.

3. Automobile Cruise Control: An Example Real-Time System

We use a variant of an automobile cruise control system, presented in [8], to illustrate the concepts developed
in this paper. Automobile cruise control is a well studied example to illustrate real-time design methods, including Octopus [3], ADARTS and CODARTS [10]. In order to keep the example manageable, we have selected only a subset of the functionality. The cruise control system presented includes a simple control loop behavior (maintaining automobile speed at a desired cruising speed), and must also respond to many external events triggered by the driver (e.g., engaging cruise control, pressing the brake, etc.). Thus, it includes both time-driven and event-driven behaviors.

The primary function of the cruise control system is to perform automated speed control, which is achieved through a closed-loop feedback control system. This is used whenever the driver sets the car into “cruising” mode, and maintains the speed of the car to the desired cruising speed. Feedback control is also used to accelerate or decelerate to a memorized cruising speed, when the driver wants to resume cruising. In either case, the closed loop control is triggered by a periodic timeout, and the processing involves (1) determining the current speed of the car, and (2) updating the throttle value based on the current speed and the desired speed. A subsidiary function involves keeping track of the current speed of the car, and continually updating the speedometer display.

In addition to maintaining the speed of the car, the system must respond to real-world events triggered by the driver, for example, pressing the brake pedal, or turning the cruise control lever to cruising position. The response of the system depends on the internal state of the system, and the behavior of the system can be specified using a state machine.

### 3.1. ROOM Model

We have created a ROOM model by first developing actors that serve as hardware wrappers for the input and output devices. The main functionality of the system is embodied in a cruiseControl actor. The actors and their interconnection are shown in Figure 2. As can be seen in the figure, the cruiseControl actor interacts with all other actors, through its different interfaces. We have not shown here the interaction of the system with the external world, which is done through system service access points (SAPs) using the services provided by an underlying ROOM virtual machine. In particular, we will use the timing service provided by the ROOM virtual machine to trigger periodic activities [19]. Also, we will assume that the ROOM virtual machine will handle device interrupts and send them as messages to the hardware wrappers.

![Figure 2. Cruise control system’s actor structure](image)

The behavior of most of the actors is relatively simple. The brake, accelerator, and the lever actors have only a single state. They receive a message from the underlying ROOM virtual machine when an interrupt is generated for the corresponding device. The actors are responsible for passing the appropriate message to the cruiseControl actor. For example, the brake actor will send a message brakePressed, or brakeReleased to the cruise control actor, depending on the input. The throttle actor has a single input port on which it receives a throttleValue message, from the cruiseControl actor, and sends an update throttle command to the throttle device.

The speedometer actor is responsible for (1) determining and displaying the current speed of the automobile, and (2) **These actors are redundant in the system presented since their role is to simply relay the message received from the underlying ROOM virtual machine. However, in the context of a larger system, which includes, for example, an anti-lock brake system, they may be necessary and may incorporate additional functionality.**
providing the current speed of the automobile for feedback control. Accordingly, it receives an external interrupt event (encapsulated in a message) from the drive shaft, which signals one rotation of the drive shaft. The speedometer actor calculates the speed of the automobile periodically. The speed calculation is triggered by a periodic timeout message, upon which the new speed calculation is done based on the number of shaft rotations in the previous measuring period. This new speed is then sent to the external speedometer device. The speedometer actor also receives a speedRequest message from the cruiseControl actor, and returns the current speed of the car in a speedValue message. Thus, the speedometer actor has multiple interfaces, through which it receives different messages and responds to them. Its behavior is not state dependent, and thus only includes a single state, with all transitions originating from and terminating in the same state.

### 3.2. Description of the cruiseControl actor

The cruiseControl actor synchronizes cruise control system activities which are initially triggered by the arrival of an external message, i.e., brake is pressed, cruise control is engaged, etc. The behavior of the cruiseControl actor is specified using a hierarchical finite state machine expressed using ROOMcharts. At the top level, its behavior is characterized by a finite state machine with two (composite) states – ManualControl, representing that the car speed is under manual (driver) control, and AutomaticControl, representing that cruise control is engaged and the car speed is under automatic control. Figure 3 depicts the transitions between the two states. As can be seen, the system initially starts in ManualControl state, and the speed of the car is under driver’s control. The cruise control system is switched to AutomaticControl state when the driver shifts the cruise control lever to either cruise or resume position. Automatic control can be stopped by the driver by pressing the brake or accelerator, or by explicitly turning cruise control off.

Figure 4 shows the behavior of the car under automatic control. In the Resuming state, the car automatically accelerates or decelerates to the last memorized cruising speed, and then switches to the Cruising state, in which the desired cruising speed is maintained. The decomposition of these two states is similar and shows the operation of the feedback control loop, triggered by a periodic timeout message.

![Figure 3. Top Level Behavior of cruiseControl actor](image)

The cruiseControl actor synchronizes cruise control system activities which are initially triggered by the arrival of an external message, i.e., brake is pressed, cruise control is engaged, etc. The behavior of the cruiseControl actor is specified using a hierarchical finite state machine expressed using ROOMcharts. At the top level, its behavior is characterized by a finite state machine with two (composite) states – ManualControl, representing that the car speed is under manual (driver) control, and AutomaticControl, representing that cruise control is engaged and the car speed is under automatic control. Figure 3 depicts the transitions between the two states. As can be seen, the system initially starts in ManualControl state, and the speed of the car is under driver’s control. The cruise control system is switched to AutomaticControl state when the driver shifts the cruise control lever to either cruise or resume position. Automatic control can be stopped by the driver by pressing the brake or accelerator, or by explicitly turning cruise control off.

In the ManualControl state, the speed of the car is under driver control, and the feedback control-loop for cruise control is inactive. We assume that the periodic timers set for feedback control are turned off when the state switches to ManualControl. The ManualControl state itself is decomposed into a number of sub-states to keep track of whether the driver is accelerating, or braking, etc. Figure 5 shows the state machine describing the behavior of the system in the ManualControl state.

### 3.3. Transactions and Timing Constraints

In the cruise control system there are a number of transactions, which we have listed in Table 1 along with the timing constraints. A description of the transactions is given below.

**Shaft Interrupt and Determine Speed.** The first two transactions are used to keep track of the current speed of the car. A shaft interrupt arrives every rotation of the wheels, and may arrive at the maximum frequency of 6000 Hz, giving a
Table 1. Cruise Control Transactions and their Timing Constraints

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Stimulus</th>
<th>Response</th>
<th>Period</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaft Interrupt</td>
<td>shaftInterrupt</td>
<td>-</td>
<td>(min) 10ms</td>
<td>10ms</td>
</tr>
<tr>
<td>Determine Speed</td>
<td>timeout</td>
<td>speedValue</td>
<td>50ms</td>
<td>10ms</td>
</tr>
<tr>
<td>Control Loop</td>
<td>timeout</td>
<td>throttleValue</td>
<td>100ms</td>
<td>100ms</td>
</tr>
<tr>
<td>Enter Cruise</td>
<td>cruise</td>
<td>throttleValue</td>
<td>-</td>
<td>200ms</td>
</tr>
<tr>
<td>Resume Cruise</td>
<td>resume</td>
<td>throttleValue</td>
<td>-</td>
<td>200ms</td>
</tr>
<tr>
<td>Accel Released</td>
<td>accelReleased</td>
<td>throttleValue</td>
<td>-</td>
<td>200ms</td>
</tr>
<tr>
<td>Brake Pressed</td>
<td>brakePressed</td>
<td>-</td>
<td>-</td>
<td>50ms</td>
</tr>
<tr>
<td>Accel Pressed</td>
<td>accelPressed</td>
<td>-</td>
<td>-</td>
<td>150ms</td>
</tr>
<tr>
<td>Cruise Off</td>
<td>cruiseOff</td>
<td>-</td>
<td>-</td>
<td>100ms</td>
</tr>
</tbody>
</table>

period (and deadline) of 10 ms for the transaction. The current speed is calculated every 50 ms, but has a tight deadline of 10 ms, so as to count the number of rotations accurately.

Closed-Loop Feedback Control. These transactions take place when the system is in AutomaticControl state. Figure 1 illustrates the message passing sequence for such a transaction (assuming that the cruiseControl actor is in Cruising state). Here, the transaction is triggered by a periodic timeout message sent to the cruiseControl actor from the timing service. The cruiseControl actor sends a message speedRequest to the speedometer actor, which returns the current speed in a speedValue message. The cruiseControl actor then computes the throttle value by applying the control law, and sends a throttleValue message to the throttle actor, which outputs the updated value to the external device. Both transactions have an activation period of 100ms, and a deadline of 100ms, as well.

Entering and Preparing for Automatic Control. There are several transactions which mark a state change from ManualControl state to AutomaticControl state, and have similar behavior and requirements. For example, when the lever is set to cruise position, a cruise message is sent to the cruise control actor, which makes a transition to AutomaticControl.Cruising.Ready state, and sends a message to the speedometer actor to get the current speed. When the speed is returned, the cruiseControl actor updates the throttle. When the throttle actor sends the updated throttleValue to the throttle device, the transaction is completed, and we say that the cruise control is active. Figure 6 depicts the message sequence chart for the Enter Cruise (EC) transaction. Very similar transactions take place when the accelerator is released in the ManualControl.Accelerating state or when the lever is switched to resume position. All these transactions are aperiodic, and therefore no activation rate is specified. The deadline on these transactions is specified as 200ms.

Exiting Automatic Cruise Control. Finally, there are several transactions which exit the system from AutomaticControl state to the ManualControl state. These transactions are triggered by pressing of the brake, switching the lever to off, and pressing the accelerator pedal. For example, consider the Brake Pressed transaction, which is triggered by a brakePressed message to the brake actor, which gets relayed to the cruiseControl actor. The cruiseControl actor makes transition out of the AutomaticControl state to the ManualControl state, which marks the end of the transaction, since the automobile’s speed is no longer under automatic control. All of these transactions are aperiodic event triggered, and have no specified arrival rate. The deadline for the Brake Pressed transaction is very tight and equals 50ms since it reflects an urgent stopping of cruise control. The deadline...
for the transaction arising from lever being switched to off position is 100ms, and for the transaction when the accelerator is pressed is 150ms. Figure 7 depicts the message sequence chart for the Brake Pressed transaction.

![Figure 7. Annotated Message Sequence Chart for Brake Pressed (BP) Transaction](image)

### 4. Design and Implementation Guidelines

In this section we develop some guidelines that help us apply the real-time scheduling theory results to ROOM models. The biggest challenge in applying scheduling theory to ROOM models comes from various sources of priority inversions that can result in large and possibly unbounded blocking times. In ROOM models, priorities are defined at two levels: message priorities and thread priorities. Recall that a ROOM model is, in general, implemented as a multi-threaded executable, where each thread implements one or more actors. The operating system is responsible for scheduling of threads. We assume that the operating system uses pre-emptive priority scheduling, and allows application control over thread priorities. The run-time system implementing the ROOM virtual machine is responsible for scheduling the processing of events within a single thread, and would normally employ priority scheduling based on event priorities. Thus, there are two levels of priority scheduling:

- Within the context of a single thread, the processing of events takes place in message/event priority order. This ordering is enforced by the message handling loop provided by the run-time system.
- Across the whole system, the operating system schedules the threads in thread priority order.

We take the view that, from the perspective of an application, the message priorities are the “real priorities,” and thread priorities are artifacts of implementation. Thus, ideally, processing across the whole system should be driven by the message priorities, and there is a priority inversion whenever a higher priority message processing is blocked by lower priority message processing (whether in the same thread or another thread).

In order to help dealing with priorities, priority inversion, and blocking times, we develop the notion of a virtual task. Let $e_i$ be any event/message type in a ROOM model. Recall that in ROOM, events and messages are essentially synonymous concepts, and an event is generated when a message is sent by an actor (or the ROOM virtual machine) through one of its interface components. With each such event type, we associate a virtual task $\tau_i$ that carries out the processing associated with the event. A virtual task becomes runnable when an instance of $e_i$ arrives at the receiving actor’s port, and terminates when the processing associated with the event is completed, including all the responses/events it generates. The notion of a virtual task is purely conceptual and, contrary to the generally implied connotation, a task does not have an independent thread of control. Also, the task concept abstracts away the state of an actor, and hence the actual processing time of a task will vary significantly depending on the state the actor is in when the message is processed.

Each task $\tau_i$ in a ROOM model, as defined above, has a priority defined by its corresponding event type $e_i$. Thus, blocking occurs whenever a lower priority task runs in preference to a higher priority task. Assuming that priorities to virtual tasks have been appropriately assigned, it is imperative that we be able to limit priority inversions so that the blocking time experienced due to lower priority virtual tasks is small and bounded. Furthermore, the blocking time bound must not be “brittle,” in the sense that it should be more or less independent of the number and the specific resource needs of lower priority virtual tasks. Ideally, adding some new functionality at a lower priority should not make previous analysis useless. However, this is not always possible when resources are shared. But, as we shall see shortly, it is possible to ensure that the blocking times are not affected, so long as certain restrictions are enforced when adding new functionality. There are three sources of priority inversions which contribute to blocking time. In the following subsections, we consider these sources of priority inversions and suggest design and implementation guidelines to minimize their adverse effects.

#### 4.1. Blocking due to Two Level Scheduling

The first potential source of blocking comes from inappropriate thread priorities. Recall that each actor, and hence each thread, may be receiving multiple messages at different priorities. Therefore, a fixed priority assignment to threads will inevitably result in priority inversions. Hence, we sug-
gest that the thread priority should be a dynamic attribute determined by the messages waiting to be handled, which is stated in our first implementation guideline.

Guideline 1: A thread should have a dynamic priority, which is equal to the highest priority pending message (including the message being currently processed, if any).

Following this guideline implies that message send operations should change the priority of the receiver thread, if the message being sent has a priority higher than the receiver thread’s current priority. Also, when the processing for an event is completed, its priority should return to that of the highest priority message remaining in its incoming queues. When a run-time system follows this guideline, an application can be sure that no blocking occurs due to inappropriate thread priorities. In fact, the application designer does not need to worry about the thread priorities – it is automatically taken care of by the run-time system. We note that there is an overhead associated with dynamic priority changes, since it involves making operating system calls to change priority, and that this overhead must be suitably accounted. However, recent performance numbers suggest that such system calls can be executed very cheaply, with execution times as low as a 10 microseconds or less [16, 2].

4.2. Blocking due to Message Passing

The second source of blocking comes from priority inversions arising from access to shared data structures. In the case of ROOM models, actors communicate via message passing and there is no provision for sharing data structures. However, the message passing itself requires the sharing of message queues, and therefore, the processing due to sending and receiving messages can incur blocking. Priority Inheritance and Priority Ceiling protocols [22] can be used to bound the blocking time associated with such priority inversion. We propose the use of a simpler Immediate Priority Ceiling Inheritance protocol [6], and show that using this protocol, such blocking may be greatly reduced and even totally eliminated.

The protocol works as follows: let the ceiling priority of a message queue be the priority of the maximum priority task that can access it. When a task accesses a message queue, its priority is raised to the level of the ceiling priority. The priority returns to its previous value when the access is completed. In addition to ensuring mutual exclusion (no explicit locks are needed), the protocol ensures that a task can only be blocked once, and only when it is released. When a task is released, a lower priority task may be executing with a ceiling priority of equal or higher priority. The task will then have to wait for the lower priority task to finish before it may execute. Based on the above discussion, we suggest the following implementation guideline for message passing implementation.

Guideline 2: Message queues should be implemented on a per-thread basis, with one queue for each priority level. Each such queue has a ceiling priority which is maximum of the queue priority level, and the priority of tasks that may send messages to this queue. The Send and Receive operations should be performed at the ceiling priority of the queue. No explicit locks are needed in either the Send or the Receive operations.

This guideline overrides Guideline 1 in determining the priority at which the Send and Receive operations are performed in a task. Thus, the sender will boost its priority up during a send operation when it accesses a queue with a higher ceiling priority. Likewise, if a task receives messages from a higher priority task, then the reception of the messages (i.e., the retrieval of the message from the shared queue) by the receiver will be done at the ceiling priority of the queue, and not at the message priority itself. Note that this priority change takes place only when the actual Receive operation begins, and the priority of the receiver equals the priority of the message both before and after the actual Receive.

Blocking due to message passing occurs only when either (1) a higher priority task sends a lower priority message, or (2) a lower priority task sends a higher priority message. This is important to keep in mind when determining message priorities. It is also important to ensure that any blocking due to message passing is bounded, and this is presented as an explicit guideline.

Guideline 3: The execution time for Send and Receive operations must be kept within a specified bound determined by the time-scales of the timing constraints of the system.

4.3. Blocking due to Run-to-Completion Semantics

A final source of blocking is due to the run-to-completion semantics of ROOM models. Due to the run-to-completion semantics, the processing of a virtual task may be delayed if a previous lower priority virtual task within the same thread has not finished. This blocking is inherent in ROOM execution semantics, and therefore cannot be avoided. However, the scope of this blocking is limited to within the actors mapped to a thread, and equals the maximum of the processing times associated with all the actor transitions. This is given as follows:

\[ B^{RTC} \]
task $\tau_j$.

While it impossible to completely eliminate run-to-completion blocking, it is quite possible to minimize its adverse effects. Our first guideline makes explicit the rules with which new functionality may be added to an actor, without affecting the schedulability of the higher priority transactions flowing through an actor.

**Guideline 4:** Within the scope of an actor, an upper bound on tolerable blocking time for each time-critical event should be specified. The execution times of a transition must not exceed the blocking time bound on any higher priority event. If necessary, a transition may be divided into multiple transitions to meet this bound.

Complementing the above guideline is the following guideline on how to restrict the grouping of actors into threads. It is advantageous to have as few threads as possible since many resources (e.g., message queues at the level of ROOM, and execution stack etc. at the level of the operating system) are allocated on a per-thread basis.

**Guideline 5:** Two actors may be safely grouped into a single thread if the tasks in both actors satisfy the blocking time bounds of the other actor.

### 4.4. Assigning Message Priorities

The above discussion has presented guidelines based on the assumption that message priorities are pre-assigned. In reality, message priorities are themselves artifacts of system design, and should be derived in a systematic manner from the system requirements. Since system behavior and the timing constraints are described in terms of transactions, it makes sense to first assign priorities to transactions, and then let each message have a priority equal to the highest priority transaction it belongs. A number of factors must be considered in determining message priorities, and these are outlined below.

First, since scheduling is determined by message priorities, they have a direct impact on response times, and hence the meeting of deadlines of a transaction. Thus, any assignment of transaction/message priorities must take transaction deadlines into consideration. A good heuristic is a deadline monotonic ordering for priorities.

Second, message priorities may be used to avoid unnecessary processing. For example, in automobile cruise control, if the driver switches cruise control off, then the control loop executions will be abrogated. By assigning higher priority to the CruiseOff transaction, (even though it has a looser deadline) we can abrogate the control loop faster, and reduce the response time for the CruiseOff transaction.

Finally, message priority assignment also impacts the blocking times and the overheads in the system. Consider a transaction with several tasks mapped to the same thread.

In this case, if the priority of each task is no lower than its predecessor in the transaction sequence, then for each thread, a run-to-completion blocking can take place only once. For example, let $\Gamma = \langle \tau_1, \tau_2, \ldots, \tau_3 \rangle$ be a transaction, and let $\tau_1, \tau_2$ be mapped to the same thread. Then, $\tau_1$ may face a run-to-completion blocking when it is released. However, once $\tau_1$ starts executing, and until $\tau_1$ completes, the system could only be running tasks at priority equal to or higher than $\tau_3$’s priority. That means, that when $\tau_3$ becomes runnable, no other lower priority task in the thread could be executing. This observation suggests that it is advantageous to group actors into a single thread whenever it is safe to do so in accordance with Guideline 5.

A final point worth noting is that while priorities are necessary to obtain differential treatment of tasks by the scheduler, they also introduce additional overheads in the system, especially if thread priorities are adjusted in accordance with Guidelines 1 and 2. Also, blocking due to message takes place only when priorities change within a transaction.

### 5. Schedulability analysis

In this section, we present our approach to schedulability analysis of ROOM models. Once blocking times have been analyzed, calculating the response time for transactions can be done according to real-time scheduling theory. We use the strategy shown in [24, 6], based on the following basic equation to calculate the response time of a task:

$$R_i = \sum_{\tau_j \in hp(i)} (\left\lfloor \frac{R_{\tau_j}}{T_{\tau_j}} \right\rfloor * C_j) + C_i + B_i$$

where, $R_i$, $C_i$, and $B_i$ are the response time, execution time, and the blocking time for task $\tau_i$ respectively, and $hp(i)$ is the set of tasks with priority higher than $\tau_i$. The basic idea is that the response time of a task includes the interference from the higher priority tasks (given by the first term), its own execution time, and the blocking time from lower priority tasks. Note that since $R_i$ occurs on both side of the equation, the equation is solved iteratively for the smallest value or $R_i$ that satisfies it. The response time analysis becomes more complicated when there are multiple tasks in a transaction (as is the case for us), or when deadlines exceed activation period. The reader is referred to [6, 24] for more details.

### 5.1. Cruise Control Example: Revisited

We now return to the schedulability analysis for cruise control example presented earlier. We will ignore the overheads due to the ROOM virtual machine as well as operating system overheads, although, in any real system they must clearly be taken into account. To illustrate the schedulability
Response Time Analysis. We now proceed with the response time analysis for the different transactions. In computing the blocking times and the interference from higher priority tasks, we take into account the enabling conditions for a transaction. For example, even though transactions EC, RC, and AR, which mark entry into AutomaticControl are lower priority than the control loop transaction (CL), they cannot get preempted by CL, since the system must be in ManualControl state for them to be triggered, and then CL cannot be active.

Shaft Interrupt and Determine Speed. Since the Shaft Interrupt (SI) transaction is the highest priority, its response time is easy to calculate, and equals 5 ms (2 ms execution time, and 3 ms blocking time). Next consider the Determine Speed (DS) transaction. Here again there is a single task \( \tau_{\text{timeout}} \) in the speedometer actor. Its response time can be calculated from the following equation:

\[
R = \left(\left\lceil \frac{R}{10} \right\rceil \ast 2 \right) + \left(\left\lceil \frac{R}{50} \right\rceil \ast 3 \right) + (17) + (8) = 8
\]

where, the first term is the interference from the higher priority task (SI), the second term is its own execution time, and the final term is the run-to-completion blocking.

Closed-Loop Feedback Control. We now turn attention to the control loop transaction (CL). We only need to consider preemption from SI and DS transactions, even though the exiting transactions (BP, AP, and CO) are higher priority, since any of those tasks will abrogate the CL transaction.

There are four tasks in the transaction as shown in Figure 1, and their execution times are bounded by (according to Table 2) 2, 3, 10, and 2 respectively. Thus, the total execution time is 17 ms. The total blocking time is bounded by 5 ms in \( \text{Thread}_1 \), and 3 ms in \( \text{Thread}_2 \). Thus, the response time is given by:

\[
R = \left(\left\lceil \frac{R}{10} \right\rceil \ast 2 \right) + \left(\left\lceil \frac{R}{50} \right\rceil \ast 3 \right) + (17) + (8) = 36
\]

where the first two terms are interference from the SI and DS transactions respectively.

Entering and Preparing for Automatic Control. These three transactions (EC, RC, and AR) have identical behavior, and have an execution time of 22 ms from 5 tasks. The blocking time is 5 ms within the \( \text{Thread}_1 \) (since CL cannot be active), and 3 ms within \( \text{Thread}_2 \). Thus, we get their response time of 43 ms from:

\[
R = \left(\left\lceil \frac{R}{10} \right\rceil \ast 2 \right) + \left(\left\lceil \frac{R}{50} \right\rceil \ast 3 \right) + (22) + (8) = 43
\]

Exiting Automatic Cruise Control. Finally, we look at the exiting tasks BP, AP, and CO, each of them has two tasks, with a total execution time of 7 ms. The blocking time for each of them is 10 ms, since both tasks belong to \( \text{Thread}_1 \). The response times are determined by the following:

\[
R_{BP} = \left[ \left\lceil \frac{R}{10} \right\rceil \ast 2 + \left[ \frac{R}{100} \right] \ast 7 \right] + 10 = 26
\]

\[
R_{CO} = \left[ \left\lceil \frac{R}{10} \right\rceil \ast 2 + \left[ \frac{R}{100} \right] \ast (3 + 7) \right] + 7 + 10 = 35
\]

\[
R_{AP} = \left[ \left\lceil \frac{R}{10} \right\rceil \ast 2 + \left[ \frac{R}{100} \right] \ast 10 \right] + \left[ \frac{R}{100} \right] \ast 7 + 7 + 10 = 44
\]
### Table 3. Schedulability Analysis for Cruise Control Transactions

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Period</th>
<th>Deadline</th>
<th>Priority</th>
<th>Execution</th>
<th>Blocking</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>(min) 10</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>DS</td>
<td>50</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>CL</td>
<td>100</td>
<td>100</td>
<td>6</td>
<td>17 (2 + 3 + 10 + 2)</td>
<td>8 (5 + 3)</td>
<td>36</td>
</tr>
<tr>
<td>EC</td>
<td>-</td>
<td>200</td>
<td>7</td>
<td>22 (2 + 5 + 3 + 10 + 2)</td>
<td>8 (5 + 3)</td>
<td>43</td>
</tr>
<tr>
<td>RC</td>
<td>-</td>
<td>200</td>
<td>7</td>
<td>22 (2 + 5 + 3 + 10 + 2)</td>
<td>8 (5 + 3)</td>
<td>43</td>
</tr>
<tr>
<td>AR</td>
<td>-</td>
<td>200</td>
<td>7</td>
<td>22 (2 + 5 + 3 + 10 + 2)</td>
<td>8 (5 + 3)</td>
<td>43</td>
</tr>
<tr>
<td>BP</td>
<td>-</td>
<td>50</td>
<td>3</td>
<td>7 (2 + 5)</td>
<td>10</td>
<td>26</td>
</tr>
<tr>
<td>CO</td>
<td>-</td>
<td>100</td>
<td>4</td>
<td>7 (2 + 5)</td>
<td>10</td>
<td>35</td>
</tr>
<tr>
<td>AP</td>
<td>-</td>
<td>150</td>
<td>5</td>
<td>7 (2 + 5)</td>
<td>10</td>
<td>44</td>
</tr>
</tbody>
</table>

#### 5.2. Discussion

As the cruise control example illustrates, once the priorities are assigned, it is relatively straightforward to apply real-time scheduling results in computing response times for the transactions. Thus, the full benefits of real-time scheduling theory are applicable to systems designed under ROOM following the guidelines presented. This is not to say that all systems designed in ROOM can be subjected to schedulability analysis; we have not looked at all features of ROOM, and it is possible that some of them will result in unbounded execution times. For example, ROOM allows facilities such as dynamic actor creation etc., which can lead to large priority inversions, and even potentially unbounded execution times. It is clearly important to subject the ROOM virtual machine implementation to rigorous execution time analysis before schedulability analysis can be applied to ROOM models.

The biggest difficulty comes up in assigning event priorities and determining which higher priority transactions to consider in the response time analysis. This process can get tedious in a complex system with large number of transactions. Also, we believe that it is virtually impossible to fully automate this process. One solution is to use a conservative approach, but that may yield very pessimistic results. We believe that good tool support is needed to help a designer assign appropriate priorities to transactions and to identify the higher priority transactions that must be considered for response time analysis of a transaction. The exact nature of how such a tool should be built is the subject of ongoing investigation.

#### 6 Concluding Remarks

In this paper we have presented our experience in applying real-time scheduling theory to embedded control systems developed using ROOM methodology. We used a variant of the well studied cruise control system to illustrate how the scheduling theory may be applied. ROOM is a modeling methodology originating from the telecommunications community, and is particularly suited for modeling real-time reactive systems. Furthermore, through the use of executable models and automatic code generation, it claims to eliminate error-prone discontinuities in system development. With many real-time embedded control systems being implemented in software, and with the increasing complexity associated with the control flow part of the system, ROOM becomes an attractive methodology to develop such systems. However, it has no support for the expression of timing constraints, or for applying the advances in real-time scheduling theory which are now considered critical to analyze the temporal behavior of embedded real-time systems.

The biggest challenge in applying ROOM methodology to such systems comes from priority inversions. Indeed, our investigation shows that unless proper precautions are taken, such blocking times can be arbitrarily large, and potentially unbounded. In this paper we have shown how timing constraints may be specified in ROOM, and how such timing constraints may be subjected to schedulability analysis using the real-time scheduling theory. In particular, we show how the adverse effects of blocking times can be minimized if certain guidelines are followed in the design and implementation of ROOM models and ROOM virtual machine, which provides the run-time system for systems designed using ROOM methodology. While our work was done in the context of a specific methodology, we believe the observations and guidelines are more generally applicable. In particular, the results are equally applicable to Unified Modeling Language (UML).

In this paper, we focused on relatively simple timing constraints, although, we believe that the analysis can be easily extended to include more sophisticated timing requirements. We also feel that the recent results on integration of real-time
scheduling with design of such control systems [9, 18] can also be used and applied. Also, we restricted attention to a single processor system. The applicability of ROOM to distributed systems would require that message passing in ROOM be implemented using a real-time network. We found that the biggest difficulty lies in assigning appropriate message/event priorities. While we presented some guidelines on how this should be done, it needs more careful exploration. Also, we found that many transactions cannot take place simultaneously, therefore, the blind application of response time analysis may result in unnecessarily pessimistic analysis. We believe that appropriate tool support can quickly help a designer identify the transactions that must be considered in response time analysis.

As a final remark, we are quite encouraged with our results, and think that our work is an important first step in the incorporation of real-time scheduling theory in software development tools developed to model real-time reactive behavior. We are in touch with ObjecTime personnel to carry on a more thorough investigation of the applicability of ROOM for embedded control systems. We believe that with proper tool support, and some restrictions, real-time scheduling theory may be successfully applied to systems developed using ROOM methodology, thus yielding the many benefits that it provides.

Acknowledgements. We gratefully acknowledge the help of Bran Selic of ObjecTime Ltd. who carefully read an earlier version of this paper, and gave us valuable comments that have led to an improved paper.

References