Schedulability Analysis for Automated Implementations of Real-Time Object-Oriented Models

M. Saksena* A. Ptak† P. Freedman‡ P. Rodziewicz§

Abstract

The increasing complexity of real-time software has led to a recent trend in the use of high-level modeling languages for development of real-time software. One representative example is the modeling language ROOM (real-time object-oriented modeling), which provides features such as object-orientation, state machine description of behaviors, formal semantics for executability of models, and possibility of automated code generation. However, these modeling languages largely ignore the timeliness aspect of real-time systems, and fail to provide any guidance for a designer to a priori predict and analyze temporal behavior.

In this paper we consider schedulability analysis for automated implementations of ROOM models, based on the ObjecTime toolset. This work builds on results presented in [8], where we developed some guidelines for the design and implementation of real-time object-oriented models. Using the guidelines, we have modified the run-time system library provided by the ObjecTime toolset to make it amenable to schedulability analysis. Based on the modified toolset, we show how a ROOM model can be analyzed for schedulability, taking into account the implementation overheads and structure. The analysis is validated experimentally, first using simple periodic models, and then using a large case study of a train tilting system.

1 Introduction

In a recent article entitled "The Challenges of Real-Time Software Design" [11], Selic and Ward use the words *event-driven* and *time-driven* to describe two basic 'styles' of real-time software. The time-driven style corresponds to using cyclic activities triggered by time. This software style is naturally suited for the implementation of periodic activities, such as software implementation of control-loop behavior in embedded control systems. In contrast, software written in event-driven style typically waits for an event to arrive, reacts to it by performing an appropriate computation, and then enters a dormant state waiting for the next event. The event-driven style naturally suits implementation of reactive behavior, associated with (often unpredictable) discrete events in the external system.

The time-driven style has evolved to develop methods that can predict *timeliness* aspects of a system, and is particularly suitable for systems whose performance critically depends on meeting deadlines. The predictability of event-arrivals makes it easier to do such analysis. The maturity of real-time scheduling theory [5] has led to a range of techniques, mostly based on fixed priority scheduling theory, that a designer/developer can use to perform timing and schedulability analysis of a time-driven real-time system. In contrast to time-driven software style, the event-driven software style has evolved largely to deal with the complexity arising from asynchrony, concurrency, and the inherent non-determinism due to the two. In order to deal with this complexity, various modeling languages have evolved using modeling features such as (1) object-orientation, with finite state machines to model object behavior, (2) visual modeling, with formal semantics (enabling executability of models), (3) use-cases or scenarios to model system behavior, and (4) associated CASE tools to facilitate model development, provide executability of models, as well as partially or completely automate code generation.

In reality most systems contain both time-driven and event-driven components. For example, consider computerized control systems. The control loops and other data flow paths are typical of the time-driven parts of such systems. On the other hand, such a system must also react to discrete external events such as operator commands, changes in the operating conditions, and various types of exceptions and equipment failures. In fact, most of the complexity of such a system is in this reactive component of a system. Moreover, there are timing constraints to be respected for the reactive component as well. Therefore, as systems be-

*Department of Computer Science, Concordia University, Montreal, Canada. Email: manas@cs.concordia.ca.

†Centre de recherche informatique de Montreal (CRIM), and Canadian Space Agency, Montreal, Canada. Email: aptak@crim.ca.

‡Centre de recherche informatique de Montreal (CRIM), Email: freedman@crim.ca.

§Centre de recherche informatique de Montreal (CRIM) and Concordia University, Email: rodziew@cs.concordia.ca

come increasingly complex, it is our view that one must be able to use high level modeling languages to deal with the complexity of software, and still be able to subject critical components of the system to timing and schedulability analysis.

Unfortunately, little attention has been paid to the integration of schedulability analysis into the analysis and development tools for real-time modeling languages. This is, in part, due to the very nature of event-driven software – that is the asynchrony and non-determinism. In the absence of deterministic arrival of events, it is hard to determine response times, and thus hard to analyze a system for real-time performance. On the other hand, in a recent paper [8], we showed that it is possible to perform such analysis. The key is to realize that even though a given system may be processing many events in a state-dependent manner, only a relatively small number of event-sequences are time-critical and need to be analyzed for timeliness. Thus, if we can (1) prioritize events to reflect their time-criticality, and (2) implement a system that bounds priority inversions for event-processing, then it becomes simpler to analyze the time behavior of the time-critical event-sequences. The key aspects of our guidelines presented in [8] were the need for (1) preemptability of event processing, which requires multiple threads of control performing event-processing, and (2) managing thread priorities dynamically to minimize and bound priority inversions for event-processing. In other words, threads of control and thread priorities must be viewed as artifacts of implementation for preemptability, and events and event-priorities must be viewed as design elements.

In this paper, we present our experience with the application of the guidelines presented in [8] for implementation of real-time object-oriented models. As in [8], our work uses the ROOM modeling language [9] and the ObjecTime Developer Toolset from ObjecTime Inc. The ROOM modeling language and the ObjecTime Developer Toolset are representative of the new generation of modeling and development tools for event-driven real-time software. Rhapsody from i-Logix (http://www.ilogix.com), and ObjectGeode from Verilog (http://www.verilogusa.com/og/og.htm) are other similar tools, and by and large the results presented here are equally applicable for systems developed using those tools. Also, the modeling concepts of ROOM are compatible with the Unified Modeling Language (UML) developed by a consortium of companies, including Rational, i-Logix and ObjecTime, and recently standardized by Object Management Group (OMG). Indeed, a recent joint press release from ObjecTime and Rational states that “Rational and ObjecTime are defining a comprehensive approach for the application of UML to the development of complex real-time systems.” A recent white paper [10] also describes how the ROOM modeling concepts may be derived from the general UML modeling concepts, and suggests a domain specific specialization of UML for real-time based on ROOM modeling language.

The main contribution of this paper is in presenting experimental evidence of the soundness of the ideas presented in [8] showing that it is possible to perform a priori schedulability analysis of automated implementations of real-time object-oriented software models. We have modified ObjecTime’s run-time system in accordance with the guidelines presented in [8], and then used the implementation to develop a schedulability analysis model taking into account the implementation aspects and the overheads. This model is then used to illustrate how schedulability analysis can be performed for a simple set of periodic tasks; we also validate the model through measurements. To illustrate the generality of the approach, we also present a case study of a train tilting system to illustrate to what extent the approach scales to complex ROOM models. The train tilting system modeled here represents a realistic and representative (future generation) computerized control system. Previous work on the application of ROOM/ObjecTime for the development of the train tilting system [2] had revealed serious timing problems due to the use of static thread priorities.

2 Overview of ROOM Concepts

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 [3]. 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 a 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 upto 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 architecture. A layer provides a set of services to the entities in the layer above. The linkage between layers is done at discrete contact points 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. 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.

3 Implementation of ROOM Models

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 Developer Toolset 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 [9]. The ObjecTime toolset includes a target run-time system (`TargetRTS`) [7], which is linked with the application code to provide a stand-alone 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.

3.1 Single vs Multi-Threaded Implementation

The simplest approach to implement a real-time system designed in ROOM is to cast all actors into a single thread. Thus, processing takes place in event priority order, and the only source of priority inversion is due to the *run-to-completion* semantics of the concurrency model. Therefore, the maximum blocking time by lower priority tasks is limited by the most "time-consuming" transition in the entire system. In many systems, this is perfectly tolerable, and leads to a simple, low overhead approach to implementation. This would be true, for example, in systems which are primarily reactive, and where the processing requirements are largely related to state changes.

This simple approach, however, ceases to be effective when the *time-driven* functionality is incorporated into a system designed in ROOM. In such systems, it is not surprising to find compute-intensive functions (e.g., signal processing functions), and the need for preemption becomes imperative to avoid significant priority inversions for higher priority event processing. A multiple thread configuration may then be used to manage preemption. In a multi-threaded configuration, each actor is mapped to one of the application defined threads. Each thread runs as a message handler for the actors that it controls. A thread may then be preempted by another thread depending on thread priorities and the scheduling of threads by the underlying operating system. Threads are also useful if a segment of code makes a blocking call to external functions. Note however, that a multi-threaded configuration comes with its own costs. In particular, inter-thread message passing is an order of magnitude more expensive than intra-thread message passing. Also, one must take into account the cost of preemption and synchronization when considering performance.

3.2 Multi-Threaded Implementation and Thread Priorities

In a multi-threaded implementation, 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 assume that the underlying operating system supports preemptive priority scheduling, and the thread priorities are under application control.

This two-level scheduling can result in priority inversions if thread priorities are not appropriately controlled. As noted earlier, *the processing associated with messages represents the "tasks" in the system, and accordingly, message priorities are the "real" application priorities.* Threads and thread priorities, on the other hand, are artifacts of an implementation. 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). The `TargetRTS` supplied by ObjecTime allows an application designer to choose thread priorities, but these priorities are static. This can lead to priority inversions since (in general) each actor, and hence each thread, will be processing messages of different priorities.

Still, many systems may be designed successfully, and indeed are, with this capability. One way to use it effectively would be to design a system such that the functionality of the system is partitioned into "control" actors (where most event processing is non compute-intensive), and "data-processing" actors (i.e., where processing may be compute-intensive, but state machines are simple, and the actor does not receive many different types of messages) parts. All "control" actors can then be mapped to a single "control" thread, while, other data-processing actors may be mapped to separate thread to manage preemption. A data processing actor can then be assigned a priority depending on the time criticality of the function it implements. The control thread can be assigned the highest priority (presumably, it will be handling all kinds of emergency messages, which require the tightest response times). While, there is some priority inversion, it may be acceptable if the aggregate low-priority workload on the control thread is not sufficient to cause missed deadlines in the data processing actors.

Dynamic Thread Priorities. It is not hard to imagine systems where a static thread priority approach would fail to work. The train tilting system presented in Section 5 is one such example. It may not be possible to nicely separate the control and data-processing functionalities into separate actors. Even when this is possible, there may be sufficient low priority workload in the control thread that can lead to missed deadlines for other data processing activities. This may happen, for example, because of various kinds of system monitoring and other background functions, which must be done in the control actors. Note that, while each of these system monitoring activities may be relatively small, the aggregate of all such activities (over a time interval of interest) may be large, and can cause significant priority inversion.

This problem was identified in [8], and it was suggested that the run-time system should automatically manage thread priorities to reflect the priority of pending messages for the thread. Thus, each thread can have a dynamic priority, which is increased whenever a higher priority message arrives for one of the actors managed by the thread. Likewise, the priority of a thread can be automatically adjusted to that of the highest priority queued message when the thread finished processing a message. With sufficient thread priorities, this simple scheme gives the advantage of preemptability offered by multiple threads, without the problems of priority inversions as described above. Of course, one must also take into account the costs of changing thread priorities.

3.3 Implementation of Dynamic Priority Threads in TargetRTS

We have made changes to the `TargetRTS` provided with the ObjecTime Developer Toolset to dynamically adjust thread priorities as described above. These modifications were made in consultation with the developers at ObjecTime. Here, we only briefly discuss the implementation aspects. Our implementation was carried out on Solaris operating system, using real-time priorities[1]. In Solaris, real-time priorities are fixed priorities (i.e., not changed by the kernel) and are higher than other thread/process priorities. These priorities can be changed from the application (with super-user privileges) using the `priocntl()` system call.

A thread[2] running with a real-time priority cannot be preempted by other threads running in the *time-sharing* or *system* class. Most user processes/threads run in the time-sharing class, while kernel threads run in the system class. Thus, a real-time thread in Solaris can only be preempted by other (higher-priority) real-time threads and interrupts.

The `TargetRTS` from ObjecTime uses a *controller* object to control a thread's execution. That is, each application thread has a controller object, and uses an underlying operating system thread to execute. The controller object is responsible for managing the message queues and implementing the message handling loop. In each iteration of the loop, it processes the current highest priority waiting message by invoking the behavior (Finite State Machine code) of the destination actor. The very nature of the message handling code ensures the run-to-completion semantics.

Our changes to the run-time system primarily consisted of (1) changing the threads implementation so that the threads are run in the real-time class with real-time priorities, and (2) changing the controller object implementation to dynamically adjust the priority of the underlying thread. There were a number of other minor changes made as well, such as increasing the number of priority levels, as well as instrumentation of the run-time system for timing purposes. We made minimal changes to the structure of the system such that ROOM models built with the ObjecTime toolset could be compiled with the modified run-time system.

4 Schedulability Analysis of ROOM Models

In this section we analyze ROOM model implementations and show how they may be subjected to schedulability analysis based on real-time scheduling theory [1, 5, 12]. Rather than develop a comprehensive schedulability analysis model, we just show how simple periodic tasks can be implemented and analyzed. The generalizations to the simple periodic task model take the same approach as has been done in the real-time scheduling literature.

[1] The `TargetRTS` provided by ObjecTime for Solaris does not use the real-time priorities.

[2] Threads in in ObjecTime are bound (to kernel threads) user-level threads. Such threads are termed light-weight processes in Solaris.

4.1 Implementing Periodic Activities

A ROOM model implemented using the ObjecTime toolset and using the `TargetRTS` consists of a single *TimerThread*, and one or more application threads (by default, there is a single MainThread which controls all actors). The TimerThread controls a *timer actor*, which implements the timing services provided by the ROOM virtual machine. The timer actor receives requests for timing services sent through timing SAPs, and manages the timers set up by actors. The TimerThread blocks for the length of the shortest timeout or until a new request is received. Upon return, it checks for expired timeouts, if any, and sends timeout messages (at the priority specified when the timer was set) to the appropriate destination actor.

Periodic activities can be implemented within any actor by setting a periodic timer through the timing service. The system then delivers a *timeout* message to the actor once every specified time period. Figure 1 shows the message flow for periodic timers. The figure on the left shows the normal message passing for periodic timers. For each periodic timeout, there are two messages: a *timeout* message sent from the timer actor, and an *informAt* message sent to the timer actor. Each *informAt* message contains the next timer expiry time, which is calculated based on the previous expiry time and the period of the timer. Before re-arming, a check is made for missed-timeouts (i.e., those timeouts whose expiry time has passed), and no messages are sent for the missed timeout intervals. A count of these missed timeouts can be requested by the application. The figure on the right shows the effect of a missed timeout.

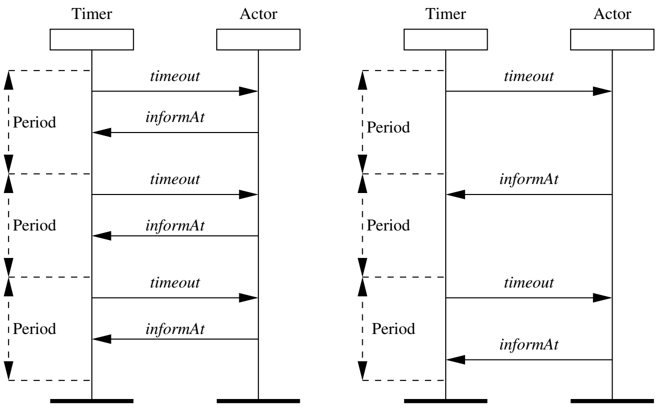

Figure 1. Message Sequence Chart(s) for Periodic Timers. The figure on left shows the normal behavior. The figure on right shows the effect of a delayed re-arming, which results in missed timeout messages.

4.2 Schedulability Analysis

In this section, we consider the timing and schedulability analysis for a set of periodic activities. We will assume that each periodic activity is implemented using a periodic timer on a separate actor. The state machine of such a periodic actor is trivial; it simply responds to the timeout messages. When the timeout message is delivered, the computation associated with the periodic activity is performed on a transition. We will further assume that the computation neither blocks nor sends a message to another actor. The basic model can easily be extended for more complex systems, as has been done in real-time scheduling theory literature [12, 5].

Let $\tau_1, \tau, \ldots, \tau_n$ be a set of periodic tasks as described above. Each task τ_i is characterized by its period T_i, and its computation time C_i. Further, let the priorities of the tasks be such that task τ_i has a higher priority than task τ_j, whenever $i < j$. Also, let Γ_i denote the thread to which a task τ_i is allocated. Then, if R_i denotes the worst-case response time for task τ_i, it can be calculated (ignoring all overheads, and assuming $R_i < T_i$) using the following equation [12].

$$R_i \; = \; \sum_{j<i} I^j(R_i) + C_i + B_i$$

where $I^j(R_i)$ represents the interference from higher priority task τ_j, and B_i represents the blocking time from lower priority tasks. Typically, through the use of priority inheritance protocols, the blocking time can be bounded. The interference from a higher priority task over a time window of W can be bounded by:

$$I^j(W) \; = \; \lceil \frac{W}{T_j} \rceil * C_j$$

Here, the first term represents the maximum number of instances of task τ_j that can arrive during an interval of length W.

The basic equations have to be modified to take into account (a) the overheads due to context switching, (b) the overheads of timeout processing, and (c) release jitter. Here we develop a simple extension of the model to take into account these factors. This model will then be used to predict response times. Figure 1 is useful to understand the additional overheads due to higher priority tasks. Recall that in our implementation, the timer actor runs in a separate TimerThread. Therefore, the message passing associated with timeouts involves inter-thread message sends, and context switches. Also, to minimize release jitter, we run the TimerThread at the highest priority, and all *informAt* messages are sent at the highest priority as well. The *timeout* messages, on the other hand, are sent at the priority requested by the application. Thus, in a simple periodic task model, each periodic actor can set up a timer with a priority equal to the task priority.

We define the following parameters to account for the implementation artifacts and overheads into our response time analysis. This is a relatively simple approximation to the overheads, but we have found that it works reasonably well in predicting response times.

- **Timeout Processing** $(C_{timeout})$: In this parameter, we lump in all the overhead associated with the processing of a timer expiry in the TimerThread. Most of this overhead is in the sending of the *timeout* message to the receiving thread, which involves accessing its message queues, as well as changing the thread's priority, if needed.
- **Re-arming Timer** (C_{rearm}): This is an even more coarse-grained parameter, and here we lump in the entire overhead of re-arming a timer. This overhead includes (a) the overhead of sending *informAt* message to the TimerThread, (b) a context switch overhead to the TimerThread, (c) the processing associated with the *informAt* message in the TimerThread, and finally (d) a context switch back to the task.
- **Context-Switch Time** (C_{cs}): This parameter is used to account for the context switch overheads.
- **Inter-Thread Message Send Time** (C_{send}): This parameter models the overheads associated with sending a message to another thread.

Interference from Higher Priority Tasks. For each higher priority task, there is an additional overhead of $C_{timeout}+C_{rearm}$, every time the timer expires for a higher priority task. In addition, we should also charge three additional context switches, one to the TimerThread, one from the TimerThread to the task, and finally another one when processing of the higher priority task is completed. Thus, the interference from higher priority tasks can be calculated as:

$$I^j(W) \;=\; \lceil \frac{W}{T_j} \rceil * (C_j + C_{timeout} + C_{rearm} + 3C_{cs})$$

Blocking Time from Lower Priority Tasks. There are several sources of blocking in this model. The first source of blocking comes from the run-to-completion semantics, which we will refer to as B_i^{RTC}, and which is given by the following equation.

$$B_i^{RTC} \;=\; \max C_j :: \Gamma_i = \Gamma_j$$

The next source of blocking comes from the processing of timer expiry for timers set for lower priority tasks. This is because, the timer expiry activates the Timer Thread, which runs at the highest priority in the system. Thus, for each lower priority task, a blocking time of $C_{timeout}+2C_{cs}$ must be added for each instance of the task that arrives within the response time interval. We do not need to take the re-arming of the timer into account, since the control must switch to the lower priority task for that to begin. We refer to this blocking as B_i^{TIMER}, and it is given by:

$$B_i^{TIMER}(W) \;=\; \sum_{j>i} \lceil \frac{W}{T_j} \rceil * C_{timeout} + 2C_{cs})$$

It is also possible that when the higher priority task is released, some low priority task is in the process of re-arming its expired timer. If the Timer thread is running, then this will delay the release of the task. We will refer to this as B_i^{REARM}, and this is bounded by the processing time associated with an *informAt* message.

Finally, there is also blocking possible due to access to shared message queues. The message queues of an actor (actually a controller) are accessed by other threads during an inter-thread send operation. Thus, a blocking is incurred if a task is activated when a lower priority task is sending a message to this thread. Using Immediate Priority Ceiling Inheritance Protocol [1], this blocking time can be bounded to the cost of one inter-thread send operation. We refer to this blocking time as: B_i^{MQ}.

In our implementation, we access shared message queues at the highest priority. This ensures that a higher priority task can be blocked by either the Timer re-arming or message queue access, but not both. Thus, the total blocking time experienced by a task is given by:

$$B_i(R_i) = B_i^{TIMER}(R_i) + B_i^{RTC} + \max(B_i^{MQ}, B_i^{REARM})$$

Response Time Calculation. Given the interference from higher priority tasks, and the blocking from lower priority tasks, it is simple to calculate the response time of a task. The computation time of a task must now include the additional overheads, and is given by:

$$C_i' = C_i + 3C_{cs} + C_{timeout} + C_{rearm}$$

Also, we must include the effect of clock release jitter (J_i), since the task's release may be delayed by the operating system due to the granularity of its system clock.

Then, the worst case response time of a task is given by the following equation (assuming $R_i < T_i$):

$$R_i(W) = \sum_{j<i} I^j(R_i) + C_i' + B_i(R_i) + J_i$$

4.3 Experimental Results

In order to validate the model, we implemented periodic models with N periodic tasks. Each periodic task was implemented as described above. We first estimated the parameters $C_{timeout}$, and C_{rearm}, and then used the estimated parameters to predict response times for tasks in a periodic task set. The predicted response times were then compared with the measured response times. Each periodic task executed a spinloop, as given below, with loopCount decided based on its desired execution time [4].

```
while (i < loopCount) { j = &i; i++;}
```

All our experiments were done on a Sun UltraSPARC-I, with a 167MHz clock, 128MB main memory, and running Solaris 2.5. Using measurements, it was estimated that 10.3 loops were required for each microsecond of execution time. For tasks of 10ms or higher execution time, this resulted in less than 1% difference in the actual measured times. All our timing was done using the `clock_gettime` call provided in Solaris (the same call is used in the `TargetRTS` for the timing services). The overhead for the clock_gettime call was measured to be between 652 and 759 nanoseconds. The clock's resolution (as provided by the `clock_getres` call) was 1 microsecond. Over a range of experiments, we found that the clock release jitter could be as high as 10 milliseconds. The values of the parameters were estimated as follows (all values are rounded up and expressed in microseconds).

$$C_{timeout} = 140,\ C_{rearm} = 405,\ C_{cs} = 135,\ C_{send} = 110$$

Here, we present the results of only one of the periodic task sets for illustration purposes. This task set consisted of 5 tasks, with periods and execution times listed in Table 1. The table also shows the predicted and the measured worst case response times. A release time jitter of 10ms was used in the calculation of predicted response time, although in this particular run, the jitter was approximately 5.5 ms.

Task	C_i	T_i	$R_i(predicted)$	$R_i(measured)$
τ_1	5	50	17.455	11.727
τ_2	12	50	29.995	23.858
τ_3	15	100	45.535	39.277
τ_4	20	100	84.975	75.431
τ_5	35	200	195.215	181.516

Table 1. Predicted and Measured Response Times for a Set of Periodic Tasks

The table shows that the measured worst-case response times fall below the predicted response times, as desired. The numbers also reveal the conservative nature of the analysis, and the overestimation was found to be largely due to (a) overestimation of the clock release jitter, and (b) some of the overheads are overestimated, since when two tasks are to be released at roughly the same time, the full overheads of timeout processing are not applicable. This is because the TimerThread processes all expired timeouts when it wakes up. In any case, the timing results reveal that in this simple case of periodic activities, the implementation of the model is quite predictable. The case study in the next section shows that this temporal determinism is preserved even in large complex systems with many actors, complex state machines, and large number of events and state dependent event processing.

5 Train Tilting System: A Case Study

In this section, we show how the analysis technique developed above can be applied to a real-life system using the example of a train tilting system. The train tilting control system is typical of computerized control systems, and includes both *time-driven* and *event-driven* behavior. Much of the software complexity of the system is in the *reactive* aspects of the system, and includes behavior related to handling operator inputs, error and limit checking, detecting alarms and failures, and failure recovery procedures. The use of ROOM modeling language, along with the visual development environment of ObjecTime Developer toolset greatly helped in the development effort. On the other hand, the system has strict timing requirements as well, both for the nominal control loop behavior as well as for reacting to external events (especially those related to failure).

Our insight into train tilting systems has evolved through our work in software development automation for embedded control systems. A preliminary ROOM model was developed at CRIM as part of a project involving the evaluation of ROOM and ObjecTime for the software development of new generation embedded control systems, in collaboration with Bombardier's Transportation Equipment Group, North America's leading mass transit company. The train tilting system described here extends that preliminary model based on our vision of how such tilting technology is evolving, with an emphasis on additional functionality. As a result, this new model does not necessarily reflect Bombardier's current or planned technologies.

5.1 Train Tilting Systems

New-generation train tilting technology allows passenger trains to travel faster on conventional (non-banked) tracks, making medium-speed train service more competitive with airlines for passenger travel [6]. By physically tilting the body of the passenger cars during the curves, the permissible train speed in the curves may be increased without exceeding the maximum allowable transient and steady state lateral acceleration experienced by the passengers. Advanced tilting systems permit train speeds in excess of 240 km/h, a significant improvement over the conventional train speed limit of 160 km/h.

Such a medium-speed train typically consists of a non-tilting locomotive pulling from two to four tilting passenger cars. Each tilting car is mounted on two trucks, with each truck composed of two axles and four wheels with a primary and secondary suspension system. Local tilting control for each car is provided by a feedback control system using servo-valves and displacement transducers on the electro-hydraulic actuators. The master tilt controller, situated in the locomotive and connected to the closed-loop tilting controllers and monitoring equipment in each car, controls the operation of all the car tilting controllers. Inertial-based instruments, situated between the primary and secondary suspension systems on the lead truck of the locomotive, provide curve-sensing data at periodic intervals. The master tilt controller calculates the tilt angles for each of the cars in the train. By placing the curve sensing subsystem on the lead truck of the locomotive, the master tilting controller has 300 milliseconds lead-time to process sensor data and calculate the tilt setpoint for the first passenger car (based on the typical locomotive length of 20 m and a train speed of 240 km/h). Each following passenger car will tilt 375 milliseconds later than the preceding one (based on the typical car length of 25 m).

The train operating environment is particularly harsh due to vibration, extreme operating temperature ranges, contamination from dirt, snow and ice, and infrequent maintenance intervals. Inertial-based sensor data is accurate but noisy. Curve data is over-sampled to allow for data filtering and rejection of erroneous measurements. Sensor, actuator and communications failures are possible; Fault-tolerance is achieved by replication of critical components and self-diagnosis at the locomotive, car and train level. On actuator failure in any car, tilting is automatically deactivated and all cars are returned to the upright position and mechanically locked in place. Robustness is achieved by self-calibration of sensors and self-test routines. The tilting master controller continually receives equipment status information via the onboard network. The train speed must be immediately reduced to the slower non-tilting speed limit in the event of a failure in any part of the tilting system.

5.2 Tilting System ROOM Model

Figure 2 shows the top-level structural view of our ROOM model of the train tilting system. The model includes actors for various sensors: one for the inertial sensor (TiltingMasterSensor) and one for speed (TrainSpeedSensor). The TiltingMasterSensor includes a CurveDataProcessor actor (not shown in the figure) for statistical filtering of the sensor data. There are actors representing the 3 passenger cars (TiltingCarControllers). The TiltingLocoConsole actor is used for operator console and display functions. It

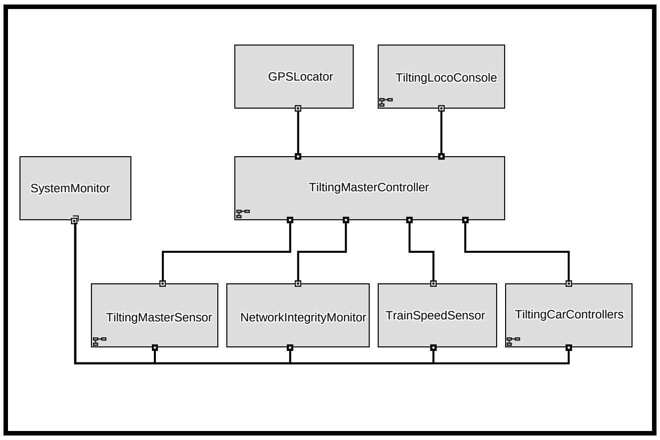

Figure 2. Top Level Structural View of the Train Tilting System

includes a DisplayProcessor actor (not shown in the figure) for processing and rendering graphical data on the console. Also, a communications network monitor (NetworkIntegrityMonitor) is used for monitoring the integrity of the network. A position reference system (GPSLocator) incorporating a Global Positioning System (GPS) receiver provides a secondary means of curve detection and allows predictive control, independent of the inertial sensing system, by locating the train's actual position on a predefined map of the train route. A train-wide system monitor (SystemMonitor) collects event logs and status information for all system components, which is used for predictive maintenance purposes. The TiltingMasterController (TMC) is the master tilting controller with interfaces to all other components. It is a hierarchical actor, and consists of two component actors, a TiltingDataProcessor (TDP) and a ModelBasedController (MBC) actor. The TiltingDataProcessor computes tilt commands based on the current speed, the inertial curve data, and the predictive curve data from the GPS system. The ModelBasedController actor takes these tilt commands and modulates them based on a model of the actuation system (feed-forward control).

The TMC behaviour is principally reactive, and consists of four principal modes of operation: Stop, Startup, Run and Fault. The controller enters the Stop state upon initialization, and it will remain in the Stop position until a START command is issued from the operator console (the TiltingLocoConsole). A START command will cause a transition to the Start-up mode, following which transitions will occur to either the Run or Fault state according to events originating in the external environment. At any time, a STOP command from the operator console will cause the TMC to return to the Stop state. A STOP command is required to recover from the Fault state.

Due to space constraints, it is impossible to give complete details of the system behavior. Instead, we focus on the behavior of the system in the nominal (RUN) state, focusing on the time-critical event sequences. These are described in the next section.

5.3 Event sequences

In the RUN mode, there are several event sequences of interest. We classify the time-critical event-sequences into two categories: (1) those that are responsible for monitoring correct operation of the system, and (2) those that constitute the main "data processing" part of the system.

5.3.1 Monitoring Event Sequences

The monitoring event sequences monitor the state of the devices and operator input. In case of failure, and/or operator command to stop the system, a quick response is needed for these event sequences to shut down the system. In the current model, we have three such monitoring event-sequences.

Onboard Network Integrity Monitoring: The onboard communications network integrity is monitored at 50 millisecond intervals by requesting a reply from the network interfaces on each passenger car. If no reply is received on any one of the interfaces, a message is sent to the TiltingMasterController. The TMC transitions to the Fault mode, disabling the tilting system, and sends a message to the OperatorDisplay in the TiltingLocoConsole.

Car Tilting Control System Status Monitoring: The car tilting control system status (including hydraulic system temperature, level and pressure and actuator displacement, and preventative maintenance information) is sent at 100 millisecond intervals to the TiltingMasterController. In the event of a fault or critical failure, the TMC transitions to the Fault mode, disabling the tilting system, and sends a message to the OperatorDisplay in the TiltingLocoConsole.

Operator Input Monitoring: The Operator input is polled once every 100 milliseconds, and any operator command is dispatched to the TiltingMasterController.

5.3.2 Data Processing Event Sequences.

The data processing event sequences constitute the main control loop functions. The control loop behavior results in sending tilt commands to the TiltingCarControllers. The tilt commands are computed based on the curve sensor values, the current speed of the train, and the predicted track curvature using the GPS position reference system. The main control loop is driven by the sampling of the curve sensor values, while the GPS and speed update are carried out asynchronously.

Curve Sensor Update: The curve sensor update event sequence constitutes the primary control loop function in the system. The event sequence is triggered by the sampling of the curve sensor values in the TiltingMasterSensor (TMS) every 50ms. The sampled values are sent to the CurveDataProcessor (CDP), grouped in a sensor data frame, which uses statistical averaging over a 5-second sliding window. We refer to this part of the event sequence as the *Curve Data Filtering* event sequence. A message sequence chart depicting the message flow for this event sequence is given in Figure 3.

The sensor is twice over-sampled to allow for rejection of one incomplete or out-of-range sensor data frame. Thus, every second filtered sensor frame (i.e., once every 100ms) is sent to the TiltingDataProcessor (TDP), which combines the TMS sensor data with the GPS location information (which is updated once per second) and the train speed (updated at 100 millisecond intervals) to calculate the new tilting setpoints for each of the passenger cars. The new tilt setpoints are then modulated by the ModelBasedController (MBC) and sent to the TiltingMasterController (TMC). The TMC formats the control value from the MBC with other control information and sends it to the individual TiltingCarControllers over the onboard network. The TCC's update their local closed-loop control setpoint based on the value received. We refer to this part of the event sequence as the *Update Tilt Angles* event-sequence.

Finally, the event sequence also includes an update of the tilt commands being sent to the TiltingLocoConsole for operator display, where a display processor actor is responsible for graphical rendering of the data on the operator's console. This part of the event sequence is referred to as the *Display Update* event-sequence.

Train Speed Sensor Update: The train speed is sampled at 100 millisecond intervals and is sent to the master tilting controller to determine the desired tilting angle and the time delay between tilting commands for each of the passenger cars. The TrainSpeedSensor samples the speed on expiry of a timer and a message containing the speed value with warning and error flags is sent to the TiltingDataProcessor. The speed sensor sets the warning flag on intermittent errors and the previous reading is resent. The speed sensor sets the error flag on repeated errors and will cause the tilting controller to disable the tilting system. In the model, the train speed is read from a data file and the timing service provides the timer expiry event.

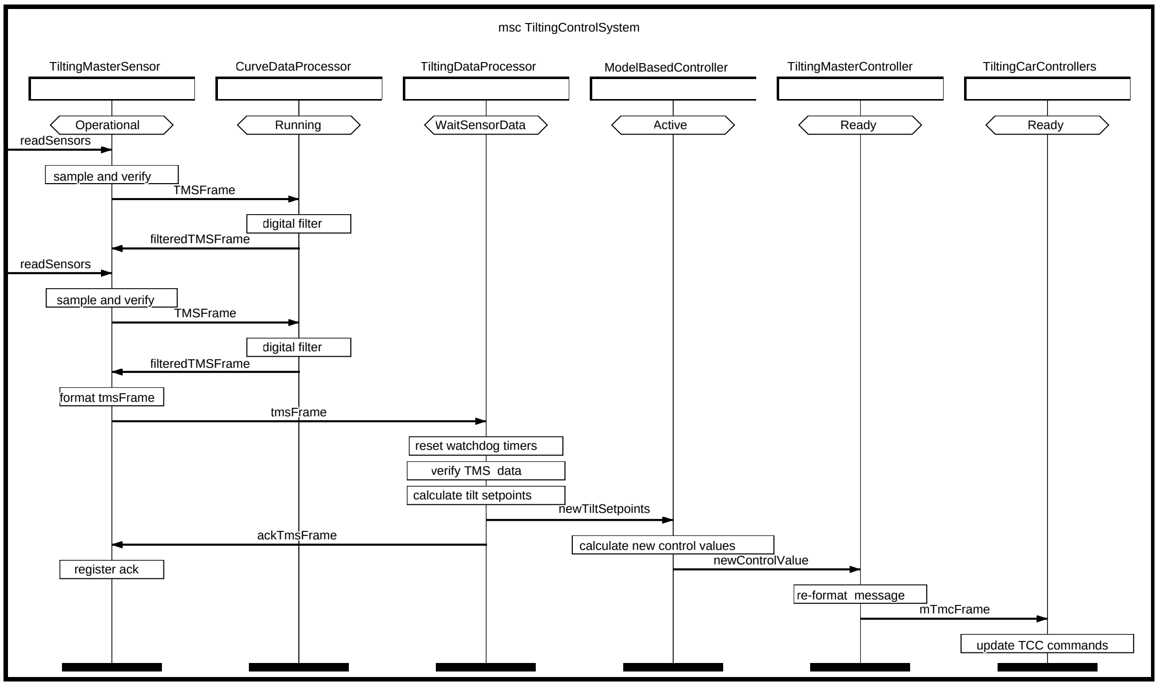

Figure 3. Message Sequence Chart for the Curve Sensor Update Event Sequence

GPS Position Reference Update: The GPS position reference system provides a secondary curve input to the tilting system, independent of the noise-prone inertial sensor, which is used to predict the required tilting angle based on a predefined database of the train route). The GPSLocator receives a position reference update at 1 second intervals from a stand-alone GPS receiver. A message containing the train position and predicted track curvature (determined from a predefined database of the train route) is sent to the TiltingMasterController. The tiltingMasterController forwards the message to the TiltingDataProcessor (where it is used to determine the tilting angle) and to the TiltingLocoConsole which forwards it to the OperatorDisplay. In the model, the behaviour is triggered by a timeout from the timing service.

5.4 Experimental Results

We conducted our experimental results with this model using curve sensor data recorded over an actual section of track between Montreal and Kingston to drive the model. This resulted in a run of approximately 100 seconds. Many of the new functionalities in the model (e.g., the GPS system, the Model Based Controller, and the Curve Data Processor) and some of the old functionality in the model had dummy code, using spin loops as used in Section 4.

Table 2 shows the assignment of priorities to the event sequences. All the monitoring event sequences were given a high priority (2) since any of them may result in stopping the control loop behavior by moving the system from RUN mode to STOP or FAULT mode. A tight response time is required in such cases. The other priorities are assigned according to the time period of the event sequence. The Display Update portion of the Curve Sensor Update event sequence is given a lower priority since this is a soft real-time task. All other events in the system were given lower priorities. Priority 1 was used for the TimerThread; this ensured that it ran whenever it was made ready.

For threads assignment, the CurveDataProcessor, GPSLocator, DisplayProcessor, and the ModelBasedController actors were each placed in separate threads, since they contained code segments with large execution times (15 ms or more). All other actors were put in one thread.

Table 2 presents the minimum, maximum, and average of the measured response times. Due to the difficulty of measuring exact execution times, we have not calculated the predicted response times. Instead, we show the approximate execution times associated with the event-sequences to give the reader an idea of expected response times. We find that, by and large, the response times are quite predictable, and as one would expect. There are a few anomalous cases, where the maximum response time for some of the high priority event sequences is significantly more than the average. A detailed examination of the timing trace revealed that this was a combination of (a) clock jitter, and (b) run-to-completion blocking due to the background system mon-

Event Sequence	**Priority**	T	C	R^{min}	R^{max}	$R^{average}$
Onboard Network Integrity Monitoring	2	50	1	2.723	18.837	3.938
Car Tilting Control System Status Monitoring	2	500	1	1.844	2.440	3.047
Operator Input Monitoring	2	100	1	2.267	5.463	2.984
Curve Sensor Update						
Curve Data Filtering	3	50	16	18.413	34.537	19.664
Update Tilt Angles	4	100	22	40.166	73.990	42.177
Display Update	7	100	25	83.295	246.207	114.265
Train Speed Sensor Update	5	100	1	19.791	24.510	20.528
GPS Position Reference Update	6	1000	51	126.506	131.541	127.257

Table 2. Event Sequences for the Train Tilting System

itoring tasks. This resulted in a net additional delay of about 15 ms, and happened only occasionally. Overall we find that the execution of the model was quite predictable, and this was revealed by hand-inspection of the detailed timing traces.

We also tested the case when the system is moved to a FAULT state due to faults in the network (detected by NetworkIntegrityMonitor) and the cars (detected by TiltingCarController), and found that due to the high priority of the events they were processed promptly and the shutdown message was received by the Tilting Car Controllers within 5-10 ms.

6 Concluding Remarks

We have presented our experience with the application of schedulability analysis to automated implementation of ROOM models. The work builds on our previous paper [8] which presented some guidelines on how priority inversions can be avoided to make schedulability analysis feasible. In this paper, we presented experimental evidence that it is indeed possible to make the temporal behavior of automated implementations of ROOM models sufficiently predictable so that a priori timing and schedulability analysis can be performed.

While our work was done in the context of a specific modeling language (ROOM) and its commercial toolset (ObjecTime Developer), the results are widely applicable. This is partly because of the standardization of the Unified Modeling Language (UML), and the fact that ROOM concepts are compatible with Real-Time UML. Moreover, other industry leaders in such tools (e.g., iLogix and Verlog) have products featuring similar modeling capabilities and code generation facilities.

We hope that the results of this work will lead to an integration of the divergent styles of real-time software development, such that in the future real-time software development can benefit from the strengths of both the event-driven and the time-driven styles through integrated toolsets which provide rich modeling capabilities on one hand, and timing and schedulability analysis tools on the other. In the future, we also hope to release the train tilting system model as a benchmark for other researchers.

References

[1] A. Burns and A. Wellings. *Real-time Systems: Specification, Verification and Analysis*, chapter Advanced Fixed Priority Scheduling, pages 32–65. Prentice Hall, 1996.

[2] P. Freedman. Investigating the suitability of objectime for the software development of embedded control systems. In *Proceedings, ObjecTime Worksop on Research in Real-Time Object-Oriented Modeling*, January 1998.

[3] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 1987.

[4] D. Katcher, H. Arakawa, and J. Strosnider. Engineering and Analysis of Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 19(9), September 1993.

[5] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, 1993.

[6] D. Lanoix. Bombardier's New Generation Tilting System. *IEEE Canadian Review*, Spring/Summer 1995.

[7] ObjecTime Limited. ObjecTime TargetRTS Guide. ObjecTime Limited, Kanata, Canada.

[8] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Contol Systems. In *Proceedings, IEEE Real-Time Systems Symposium*, 1997.

[9] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[10] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. White Paper, Published by ObjecTime, and available from www.objectime.com, March 1998.

[11] B. Selic and P. Ward. The challenges of real-time software design. *Embedded Systems Programming*, pages 66–82, October 1996.

[12] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–152, Mar. 1994.