Tool Set Implementation for Scenario-based Multithreading of UML-RT Models and Experimental Validation*

Jamison Masse, Saehwa Kim, and Seongsoo Hong
School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-742, Korea.
{jamison, ksaehwa, sshong}@redwood.snu.ac.kr.

Abstract

This paper presents our tool set implementation for scenario-based multithreading of object-oriented real-time models and an accompanying experimental validation. Our tools enable the automated, schedulability-aware implementation of real-time object-oriented models, exploiting an existing CASE tool. Our implementation is facilitated by (1) our customized run-time system modified to support scenario-based thread execution, (2) a design model template that centralizes the arrival of external inputs, (3) a model analyzer tool, and (4) a model-specific code modifier tool. Our tools simplify design by removing thread-related design concerns from the modeling process, separating design and implementation. We performed validation by conducting experiments that clearly demonstrate the performance improvements that can be gained through our scenario-based implementation: response time improvements for high priority tasks of as much as 70% and a 5-fold decrease in blocking or the elimination of blocking for some tasks.

1. Introduction

The demands of increasingly complex embedded systems make the reliability and time-to-market improvements that CASE tools can provide more and more attractive. Object-oriented CASE tools for real-time systems allow developers to take advantage of not only efficient tool-based development but also the benefits of object-oriented programming such as encapsulation, polymorphism, and inheritance. Unfortunately, the use of high-level tools often comes at the expense of speed and efficiency, two attributes of software that are often crucial in real-time systems. When using existing CASE tools, developers map objects to tasks in an ad-hoc manner in order to improve the schedulability of a model. This may involve an extended period of manually tuning both the task mapping and the design model.

In our previous work [1, 2, 3], we have proposed a systematic, schedulability-aware method of mapping object-oriented real-time models to multithreaded implementations. This was based on the notion of scenarios; a scenario is a sequence of actions that are triggered by an external input event, possibly leading to an output event [1]. In [3], we presented a multithreaded implementation architecture based on mapping scenarios to threads. This is contrary to the architecture found in current CASE tools that maps a group of objects to a thread.

In this paper, we present the details of our implementation of a set of tools that enable the automated schedulability-aware implementation of real-time object-oriented models based on our scenario-based implementation architecture. Our implementation is intended to clearly demonstrate the feasibility of practical CASE tools based on our previous work. We present an evaluation of the tools based on improvements to the modeling environment in terms of ease of use for designers. We show how our tools simplify modeling by achieving a distinct separation between design and implementation with respect to multithreading while retaining the benefits of backwards compatibility with existing tools. We also present experimental results that clearly demonstrate the performance improvements that can be gained through our scenario-based implementation as generated by our tools.

As in the previous work, we selected UML-RT [5] as our real-time object-oriented modeling language. We make use of an existing CASE tool that supports UML-RT, Rational RoseRT, as well as incorporating immediate priority inheritance protocol (IIP) [4] and preemption thresholds [6, 8, 9]. As depicted in Figure 1, the implementation is facilitated by:

- Our design model template that centralizes the arrival of external inputs for the creation of application models compatible with our tools
- Our analyzer tool that analyzes the model developed within RoseRT by parsing the generated source code to derive a new model of the application. The new model represents the system as scenarios in a tree structure that depicts the possible executions or

* The work reported in this paper is supported in part by MOST (Ministry of Science and Technology) under the National Research Laboratory (NRL) grant 2000-N-NL-01-C-136

actions of the scenarios

- Our modifier tool that modifies the source code for a specific application model. When the modified source code is compiled and linked with our customized run-time system, it generates an executable conforming to our scenario-based threading model.
- Our customized version of the RoseRT run-time system modified to support scenario-based thread execution, IIP, preemption thresholds, and non-preemptive groups

Our experimentation shows that this implementation is not only feasible for generating applications but also produces significant performance improvements.

The remainder of the paper is organized as follows. Section 2 summarizes UML-RT and explains the multithreading implementation of the RoseRT CASE tool. Section 3 presents an overview of our implementation approach for the development of our tools as opposed to that of existing tools. Section 4 presents the details of scenario-based multithreading with UML-RT and how our scenario-based run-time system was implemented. Section 5 explains how we have addressed the problems associated with generating scenario-based applications. Section 6 presents the results of our experimentation, comparing the performance of the existing tools and our scenario-based tools with the same model. The final section concludes the paper.

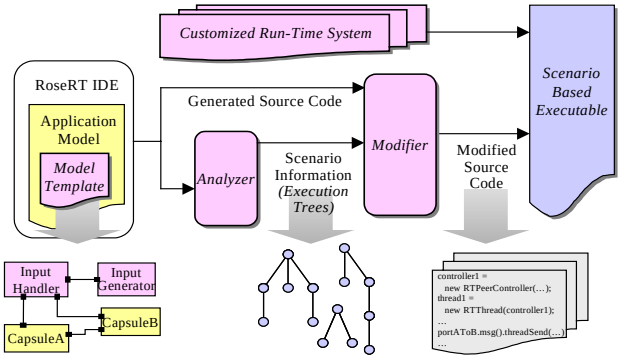


Figure 1. Scenario-based tool chain.

2. UML-RT and its CASE Tools

In this section we provide an overview of UML-RT, our chosen real-time object-oriented modeling language. We also explain one of its CASE tools, Rational RoseRT, which we exploit to implement our scenario-based implementation architecture.

2.1. UML-RT

UML-RT is a graphical language for modeling real-time applications. Rational Software developed UML-RT and the product Rational RoseRT that is used to model applications as well as generate executable applications from models. It has been developed to properly represent complex, event-driven, possibly distributed systems. UML-RT is based on the UML syntax with new constructs inspired by ROOM [10]; another object oriented modeling technique for real-time systems.

The basic element of model construction in UML-RT is a capsule. A capsule represents an object within the system and communicates with other capsules exclusively through interfaces called ports. As shown in Figure 2 (a), capsules and their connected ports compose one aspect of the model, the structure diagram that defines the communication channels within the model. A finite state machine, represented by the capsule's state diagram, represents the behavior of a capsule. An example of a state diagram is depicted in Figure 2 (b). Receiving messages via ports causes the state machine to make transitions, executing the logic contained in the capsule.

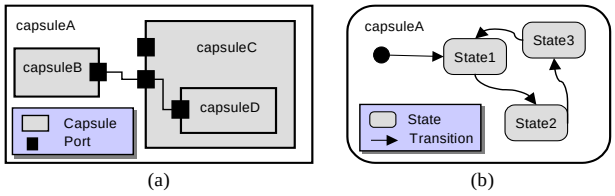


Figure 2. UML-RT (a) structure and (b) behavior diagram.

2.2. RoseRT CASE Tool

RoseRT is a CASE tool that allows users to design object-oriented real-time systems using UML-RT. Within the model, users can define all of the state machines and communication ports required for the model as well as actual code that will execute when a capsule receives messages causing the state machine to make transitions. Models created with RoseRT can be compiled and linked with the run-time system supplied with RoseRT, generating a working application directly from the model. In the generated system, objects called controllers are each assigned to a physical thread, driving execution within the run-time system. A controller takes messages to be delivered from its message queue and invokes the behavior defined by the destination capsule's finite state machine. Messages are processed according to their priorities when multiple messages are waiting in the queue.

The Rational RoseRT run-time system is configured to run on systems that support multithreading with two threads by default: one called *MainThread* that is used for model execution, and another that operates the timing service. Multiple threads can be used to execute parts of the model that *MainThread* is responsible for in the default implementation by creating a design that maps capsules, or aggregations of capsules to logical threads. Logical threads are an abstraction provided by the

RoseRT environment that are in turn mapped to physical threads that actually execute on the target platform. Every capsule instance will execute within only one thread for the duration of its lifetime. The physical threads can be assigned priorities that will cause the capsules they execute to operate at that priority.

2.2.1 Capsule-Based Multithreaded Models and Their Drawbacks

The multithreading strategy used by the capsule-based run-time system implementation, centered around a capsule to thread mapping, causes generating good multithreaded models to be more difficult than necessary. One reason for this is that multithreading within the model's proper execution, that is, delegating the work of the main thread above to two or more threads requires explicit code to be written to enable multithreaded execution. As stated in [7]:

> *If you want a capsule to run in another user-defined thread, you must incarnate that capsule in that thread at run-time. Only optional capsules may be placed on threads other than the MainThread.*

Another complicating factor in creating multithreaded models is the assignment of thread priorities within RoseRT. Individual threads can be assigned their own priorities, ensuring that all of the capsules managed by that thread execute at the desired priority. This means that developers must not only consider the best model to represent the application domain in question, but also consider the performance constraints that a given model might impose on the system.

For example, if a given set of capsules provides services required by a very performance sensitive part of the system and other services used by less important parts of the system, the developer is faced with a serious design dilemma. There are two potential solutions:

- One solution would be to move all of the capsules to a high priority thread. However, this causes all of the services provided by the capsule to become high priority ones, which means that low priority scenarios that execute several messages within these capsules create an extended priority inversion.
- Another solution, one that avoids the priority inversion problem, is to break the capsules into two, the part offering the time-sensitive service being placed in one capsule mapped to a higher priority thread and the remaining functionality mapped to a low priority thread. Rather than using message passing to communicate (which would re-introduce the problem we are solving) the two pseudo-capsules must share memory for the data required by both in some way. With each part operating on different threads this would require that thread-safe read and write access be provided by the developer to accommodate this, further complicating development.

Both of these solutions are less than satisfactory and are caused by the capsule-centric threading model. Our approach to the work of a real time system in a scenario-oriented way eliminates these design issues as well as further simplifies modeling. Priority inversion in our implementation is limited to the duration of a single transition of a lower priority scenario. In the capsule-based architecture extended priority inversion is possible since, when a low priority scenario spends its whole execution at an elevated priority it is non-preemptable for its entire execution.

3. Our Implementation Approach

In this section, we present an overview of our implementation approach that is composed of exploiting an existing case tool, adopting scenario-based multithreading architecture, and using a design template for scenario-based models.

3.1. Exploiting Existing CASE Tools

For our implementation, we have chosen to exploit an existing CASE tool, Rational RoseRT. We have selected Rational RoseRT since it is widely used and supports UML-RT. As Figure 1 illustrates, the code generated by a completed RoseRT model is used as input into our code conversion tools. The analyzer first parses the code, extracting the model information, and then the modifier alters the code, transforming the executable into a scenario-based model. This modified model code is compiled and linked with our scenario-based run-time system.

Our scenario-based run-time system was built by modifying the source code provided for the original capsule-based RoseRT run-time system. The modified run-time system is designed to support scenario-based implementations as well as the original capsule-based execution mode and thus retains backwards compatibility with the RoseRT integrated development environment.

3.2. Multithreaded Implementations using Our Scenario-Based Tools

Our scenario-based tools provide a process for developing multithreaded models with improved schedulability along and a reduction of code complexity, as the thread assignment is taken out of the UML-RT modeling process itself.

Our analysis tool operates on the RoseRT model's generated code to produce a list of threads by composing a scenario-based model of the application. The modifier alters the code according to the thread priorities and

preemption thresholds selected by developers. This approach has the flowing benefits:

- The model can be developed as if it were a single threaded implementation; no special code is required for thread management. This eliminates the mix of multithreading implementation and design that is present in the capsule-based environment.
- Schedulability is enhanced since the execution of a given thread follows the actual work done by a given scenario. Developers can be certain that all of the actions required to complete that scenario execute at the desired priority.
- Priority inversion has an upper bound of the duration of the processing of a single message by the scenario causing blocking.

3.3. A Design Model Template for Isolating Input Handling

In order to facilitate the development of scenario-based models, we have designed a simple template around a single capsule called *InputHandler*, which provides a central point of entry for external messages. Our analysis tools rely on the fact that all external messages are routed through a capsule within the model that inherits from our capsule *InputHandler*. The child capsule that is incorporated into the model need only add the communication channels into the actual application model in order to build a model around it.

4. Scenario-Based Run-Time System

In this section we present our run-time system with support for scenario-based multithreading. We present the details of how we solved the problems of (1) identifying the scenario execution model that corresponds to the UML-RT behavioral model, (2) scenario-aware message passing, ensuring that a scenario executes on the appropriate thread; (3) guaranteeing run-to-completion semantics, which ensures that the objects' state machines are properly executed.

4.1. Scenario Execution Model

Unlike the original run-time system implementation, creating a multithreaded application from a model using our run-time system can be done without writing explicit code for thread manipulation. This makes obsolete the logical and physical thread management, described in Section 2.2. Our tools provided for use with the modified run-time system manage these functions after the application is completed by manipulating the generated source code.

Threads are permitted to traverse multiple capsules, allowing the capsule's operations to function at different levels of priority depending on the scenario the capsule is participating in at the time. This creates a threading model that functions almost completely orthogonal to the UML model: the capsule relationships have little impact on the thread behavior. The thread priorities are assigned based on scenarios, the actual work they are found to do in the system, not the capsules they execute. On the other hand, this creates the environment where only one thread executes on a given capsule at a time, but many threads may execute on a capsule throughout the system lifetime. This means that writing thread-safe code for the new run-time system is essentially unchanged as long as data that is shared between capsules is guarded appropriately.

Message priorities are still supported; the executing thread processes higher priority messages first. This can be used to re-order the execution of a scenario but does not impact schedulability.

4.2. Scenario-Aware Message Passing

The most significant changes to the run-time system are the modifications that have been made to implement a scenario-aware message handling process. In order to support scenarios, the run-time system must be able to identify messages that are intended to start a new scenario and exactly which scenario the run-time system should start. For this, we extended the class used to send messages, *RTMessage*, to include fields to store the scenario information, which is the scenario id/priority, preemption threshold, and target thread id.

Normal capsule-to-capsule messages in models are sent using the *send*()*[1] methods. These methods were modified to support the passing in of scenario information using an *RTMessage* object. We have named these new implementations *threadSend*()*. Messages are also relayed through the timing service using the methods *inform*()*[2]. These methods are considered by our run-time system to always mark the beginning of a scenario and were re-implemented as *threadInform*()*. These new *thread*()* methods are not intended for direct use by developers. The calls to these methods are generated by out modification tools to be discussed later.

The scenario information required by these methods is passed through an additional parameter. Figure 3 shows a pseudo code for the scenario-aware message sending implementation. As shown, it first checks the destination scenario for the message to be delivered and delivers it to the controller in charge of executing that scenario. The method *getController()* was added to the singleton object *RTMain* to permit a controller to retrieve a target controller reference by passing in a scenario id. After acquiring a reference, any message passing method need only call the appropriate controller's *receive()* method.

---

[1] *send()* and *sendAt()*. *sendAt()* is used to index multiple ports.

[2] *informIn()* for setting single timeouts and *informEvery()* for periodic timers

The original run-time system ensures that the state machines represented in each capsule instance are thread-safe by always having one thread execute within a capsule exclusively. In order to ensure thread-safe execution in our customized run-time system, each capsule must be guarded by mutexes. The class *RTActor* was augmented with a private mutex object (*RTMutex*) to guard all capsule instances. For backwards compatibility, each capsule maintains a reference to the controller that has locked it. This locking mechanism is also important in examining the next part of our implementation, scheduling using IIP and preemption thresholds.

```
targetThread =
          RTMain::getController(msg.threadId);
// for backwards compatibility
if (targetThread == 0)
  msg.targetCasuple.targetThread->receive(msg);
// scenario-aware message passing
else
  targetThread->receive(msg);
```

Figure 3. Scenario-aware message sending implementation.

4.3. Guaranteeing Run-To-Completion Semantics

As stated earlier, each capsule in our scenario-based implementation requires an object lock to ensure thread-safe execution and preserve run-to-completion semantics. For our implementation we have selected immediate inheritance protocol (IIP). IIP ensures reliable real-time scheduling by preventing deadlock and our previous work has shown it can reduce blocking times by as much as half and also has the added benefit of reducing the number of context switches incurred during execution [4]. We implement IIP using the scheduling described in the POSIX real time standard 1003.1b [13] and 1c [14]. In order for IIP to function properly all of the mutexes that are accessed by a given thread must have priority ceilings. To accommodate this we have added this property to the mutex wrapper class *RTMutex*. Our code modifier tool assigns the ceiling of each mutex to the highest priority of the scenarios that access it since we adopted IIP with priority ceiling as in [4].

5. Scenario-Based Application Generation

The construction of an application that can benefit from scenario-based multithreading is dependent on its ability to leverage the extensions we have added to our run-time system. Here we present our tools for generating scenario-based applications. These tools, as shown in Figure 1, address the following essential issues, (1) how to easily identify scenario starting points that result from external messages in an automated way, (2) how to construct a scenario-based view of the application, and (3) how to alter the application source-code generated by RoseRT to take advantage of the scenario-based multithreading support in our run-time system.

5.1. Identifying External Scenario Initiation Points

In order to create a model that takes advantage of our scenario-based multithreading, we need a way to identify scenarios in an automated manner. To facilitate this, we have devised a model template that centralizes the external messages that trigger new scenarios. A capsule called *InputHandler* centralizes those external inputs that are injected into the model to start scenarios. Additionally, we have created a capsule for simulating the external inputs called *InputGenerator* and a class called `Signal` that is used to package the data for communication between *InputHandler* and *InputGenerator*.

Different strategies for retrieving external inputs for the model can be accommodated by making a simple change to our analyzer tool script. The discovery of external message origins can be implemented by any special means that might be required by a specific application. Any mechanism, such as interrupt handling, could be easily incorporated into this implementation.

In order to test specific sequences of input, there is also a capsule called *InputGenerator* that can be used as a test-harness. It is designed to read a file that provides the inputs into the model and pass those inputs to the *InputHandler*. The *InputGenerator* simply reads a list of inputs from a text file and packages them in *Signal* objects. This file also specifies the time for the data should be delivered to the *InputHandler* by the *InputGenerator*.

Note that no multithreading-related items need to be incorporated into the model. Multithreading is not in anyway related to the model construction, other than the need for thread-safe code if capsules are to bypass RoseRT's message passing communication facility.

5.2. Constructing a Scenario-based Model of an Application

A scenario-based model is a model that describes an application based on the execution paths taken by the scenarios it is composed of. In order to construct scenario-based models of applications, we have built an analysis tool that traces the execution and communication routes of scenarios through the source code. Using this model, we can accurately represent the application in a way that makes the information required to take advantage of our scenario-based run-time system.

The source code analyzer first makes a single pass through the source code, getting all the information about the capsules incorporated into the model. Additionally, the analyzer gathers enough information to map the ports (communication channels) between the capsules. This

port information allows trees to be formed that map out the execution pattern by tracing the communication channels with a second pass through the source code. These capsule execution trees can be linked together to form complete scenario execution trees. The root of each tree is in fact an external scenario initiation point or a timing service triggered event and every node in the tree represents either the execution of a method or passing messages between capsules. Each edge represents a possible execution path.

When the analysis completes, the information it gathers is a list of all of the possible scenarios, their starting points and a list of all the capsules accessed by each scenario. Developers need to supply both the priority as well as the preemption threshold for each scenario. The priority and threshold need to be assigned based on the work that the scenario does and the deadline imposed upon it. This process can be aided by profiling tools that calculate or estimate worst-case execution time and analyze scehdulability.

5.3. Adapting the Application Source Code for Scenario-Based Multithreading

Adapting the application source code generated by RoseRT for scenario-based multithreading involves the integration of the information provided by the analysis into the source code. Proper integration involves three modifications: (1) updating the identified scenario starting points, where external messages are introduced into the model, (2) setting the ceilings of the mutexes guarding each capsule, and (3) inserting code for thread creation and destruction.

The automated source-code modification process uses our modifier tool and the information in the output of the analysis to replace the original *send*()* and *inform*()* method calls with the appropriate call to our *thread*()* methods. Appropriate priorities, thresholds, and groups are passed into these respective calls. These changes ensure that all the scenarios found in the system are started appropriately.

The modifier must also collect the information about capsules accessed by each scenario from the output of the analyzer. Each of the capsules must be assigned a priority ceiling to ensure proper scheduling for the system. This ceiling is the maximum priority of all of the scenarios that access a given capsule. In order to actually set the priority ceilings, a call to *setPriorityCeiling()* is added to the constructor of each of the capsules in the system.

The final task for the modifier is to add the code for creating and destroying threads to the model. Our code modifier synthesizes the bodies of the thread creation and deletion functions of the *RTMain* class so that it creates all the derived physical threads. The RoseRT run-time system calls this function once during initialization.

6. Experiments and Performance Results

In this section we present an experiment based on a RoseRT model to show the performance improvements that can be achieved with our scenario-based multithreading over the capsule-based multithreading strategy. Our results clearly show an improvement in performance with respect to blocking time and also scenario response times.

In our experiments, we used the RoseRT run-time system 2001.03.00 compiled with GCC 2.95.3 using the FSU-pthread library[3]. The target environment was Sun Solaris 9 (SunOS 5.9) on a Sun Microsystems Sun Blade 1000. Our scenario-based version of the model was first constructed as a single-threaded implementation and transformed into a scenario-based implementation using the process described in this paper. The complete set of scenarios is described in Table 1, where a priority of 12 represents the highest priority task and 1, the lowest. The model was run once in single-threaded mode in order to estimate the worst-case execution times listed in Table 1. This information was used to derive the priorities using the method described in [8]. The capsule-based multithreaded implementation was also adapted from the single-threaded model but the mapping of capsules to threads was done based on the guidelines described in [11, 12].

In both implementations, there is an additional thread for operating the timer service that runs at the highest priority and can preempt any other thread. Any variance in the timer service thread in turn impacts all of the running threads' behavior as they all may incur interference, which is responsible for much of the variance we found between executions.

Table 1. Scenarios used for experimentation.

Scenario ID/ Priority	Deadline	Measured WCET	Run Count
1	3000000	7124	104
2	2000000	3848	88
3	2000000	7902	88
4	1500000	5156	72
5	1000000	3757	40
6	1000000	11401	3508
7	350000	18693	152
8	350000	29221	152
9	300000	12494	1276
10	250000	45817	2920
11	200000	6910	72
12	100000	16480	812

6.1. Performance Metrics

[3] The FSU pthread library was used in order to guarantee process-level I/O blocking, which was needed to accurately measure response times and to ensure proper scheduling.

In our experiments, we focus only on the benefits gained from our scenario-based mapping strategy in terms of schedulability. Though we do not evaluate it here, the benefits gained from using preemption threshold scheduling were discussed in other papers extensively [8, 9]. The experiments described here were conducted using only the scenario-based implementation architecture with IIP and no preemption thresholds and is compared with the capsule-based implementation. Our performance metrics used to measure schedulability are (1) blocking time and (2) response time.

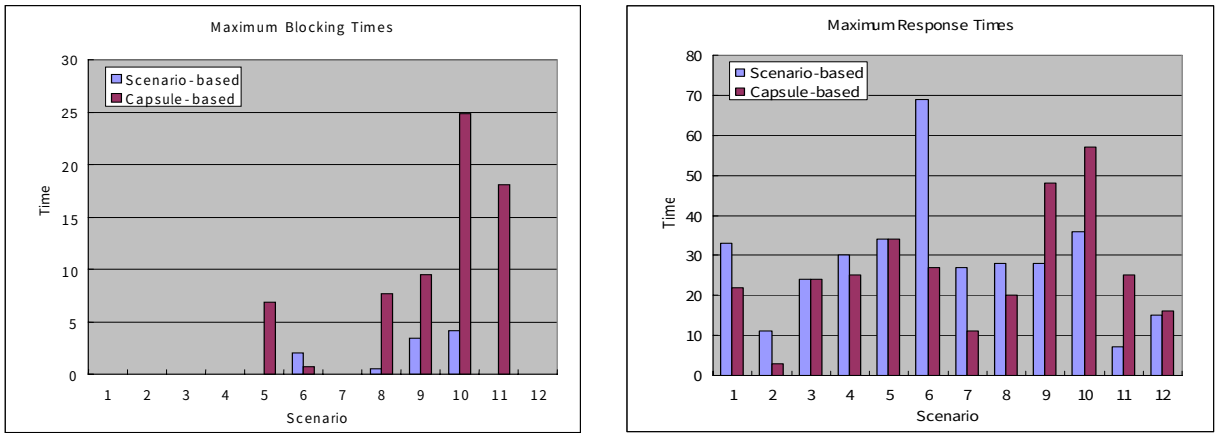
Blocking time is the duration of priority inversion for a given scenario while response time is the duration of time from the designated start time of a scenario to its completion. In the capsule-based mapping, the blocking for each scenario can occur every time messages within the scenario are passed to lower priority threads. In our scenario-based mapping, the blocking occurs only once before a scenario starts its execution when a lower priority scenario was executing while locking a mutex with a higher priority ceiling.

The response time of a scenario is composed of (1) the blocking time, (2) inter-thread message passing time, (3) the scenario execution time, and (4) the time spent preempted by higher priority scenarios. In the capsule-based mapping implementation, the inter-thread message passing can occur far more frequently than in our scenario-based mapping implementation.

6.2. Performance Results

Figure 4 (a) shows a comparison of the worst-case blocking times incurred by the two implementations. The scenario-based implementation does incur blocking but the blocking duration is significantly shorter than in the capsule-based implementation. It is also worth noting that blocking in the scenario-based model does not exceed the measured worst-case execution time of any of the critical sections that could cause these blockings. This demonstrates that our scenario-based implementation in fact produces a system with bounded blocking times, significantly improving the predictability of execution. The capsule-based mapping of threads, on the other hand, shows long blocking delays that reflect the non-preemptable nature of the capsule-based mapping. The improvement is most notable for scenario 11 where blocking is eliminated completely, and scenario 10 where blocking under the scenario-base implementation is 5 times less than the capsule-based implementation.

Figure 4 (b) shows the worst-case response time results for both the capsule-based implementation and the scenario-based implementation. Figure 4 (b) clearly shows that the response times of high priority scenarios significantly improve in the scenario-based implementation architecture. Scenario 6 can be observed to perform much worse under the scenario-based implementation with respect to maximum response time while incurring slightly more blocking. This is attributed to preemptions incurred due to higher priority tasks; the same can be said of scenarios 1, 2, 4, 7, and 8. It is important to keep in mind that response time improvement is defined by the performance of high priority tasks. We clearly demonstrate significant improvements in the three highest priority tasks that benefit from our tools with a maximum response time improvement of 42%, 37%, and 72% for tasks 9, 10, and 11, respectively. This improvement can be directly attributed to the blocking time reduction as well as the reduction in inter-thread message passing. Scenario 12 executes entirely within a single thread and does not share objects with any other scenario, thus in both implementations experiences no blocking and receives no benefit from the scenario-based architecture.



(a) (b)

Figure 4. (a) Maximum blocking times and (b) maximum response times measured during experimentation.

Over the course of experimentation, we found that models often would be schedulable with the scenario-based architecture but the same model incurs deadline misses when executed using the capsule-based architecture. This was observed when models grew in complexity, more tasks ran concurrently, and more blocking took place. At 12 scenarios, the model used in this experiment represents the least complex model that shows a significant advantage for the scenario-based architecture. This implies that the benefits of this strategy further increase as the size of a system grows. Not only does it remove the compounding complexity of manually defining thread assignment but it also benefits from a greater performance improvement as a model grows more complex.

7. Conclusion

We have presented a complete implementation of the scenario-based multithreading architecture for real-time object oriented models as well as our experimental results that validate this implementation. Our implementation successfully exploits an established UML-RT modeling tool, RoseRT, by designing a scenario-based run-time system that is not only based on the original RoseRT run-time system but maintains backwards compatibility with it.

We have developed a tool for analyzing code generated from RoseRT to construct a new application model that clearly shows developers the execution paths that are represented by the scenarios the analyzer generates. We have also presented a code modification tool that converts the non-multithreaded source code into code that can be compiled into a scenario-based multithreaded application. The result is an implementation architecture that is not only more schedulable and efficient but also easier for developers to use than the existing capsule-based multithreaded architecture. Our multithreading strategy and tools better define the division between the design and implementation of real-time systems using UML-RT. We have clearly shown that concerns about multithreading can be removed from the modeling process and doing so does not handicap the design process, but instead improves it.

Our experimental results clearly show a significant improvement in response times and a reduction in blocking times for our experimental model. We also noted that schedulability and response time improvements over the capsule-based architecture seemed to correlate with increasing complexity. These results show that this architecture is not only viable as a means to eliminate the manual thread assignment required in capsule-based architectures but also provides significant performance gains.

In the future, we will continue our research based on this implementation in the hopes of supporting distributed systems and incorporating modeling elements to support deadline miss handling. We are also considering the potential application of quality of service concepts or models to our research.

References

1. S. Kim, S. Cho, and S. Hong. Schedulability-aware mapping of real-time object-oriented models to multithreaded implementations. In Proceedings of International Conference on Real-Time Computing Systems and Applications, pp. 7-14, 2000.
2. S. Kim, S. Cho, and S. Hong. Automated implementation of distributed real-time systems using real-time object-oriented modeling, In Proceedings of IFAC Workshop on Distributed Computer Control Systems (DCCS). pp. 136-141, 2000.
3. S. Kim, S. Hong, and N. Chang. Scenario-based implementation architecture for real-time object-oriented models, In Proceedings of IEEE International Workshop on Object-oriented Real-time Dependable Systems, 2002.
4. S. Kim, S. Hong, and T.-H. Kim. Perfecting preemption threshold scheduling for object-oriented real-time system design: from the perspective of real-time synchronization, In Proceedings of ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). pp. 223-232, 2002.
5. Rational Software Corporation. http://www.rational.com.
6. W. Lamie. Preemption-Threshold, White Paper, Express Logic Inc., Available at http://www.threadx.com/wppreemption.html
7. Rational Software Corporation. Rational Rose RealTime User Guide: Revision 2001.03.00, 2000.
8. Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In Proceedings of IEEE Real-Time Computing Systems and Applications Symposium, pp. 328-335, 1999.
9. M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds, In Proceedings of IEEE Real-Time Systems Symposium, pp. 25 -34, 2000.
10. B. Selic, G. Gullekson, and P. T. Ward. Real-time object-oriented modeling. John Wesley and Sons, 1994.
11. M. Saksena, P. Freeman, and P. Rodziewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems, In Proceedings of IEEE Real-Time Systems Symposium, pp. 240-251, Dec. 1997.
12. M. Saksena, A. Ptak, P. Freeman, and P. Rodziewicz. Schedulability analysis for automated implementations of real-time object-oriented models, In Proceedings of IEEE Real-Time Systems Symposium, pp. 92-102, Dec. 1998.
13. Institute for Electrical and Electronic Engineers. IEEE Std. 1003.1b-1993 POSIX Part 1: System Application Program Interface-Amendment: Realtime Extension, 1993.
14. Institute for Electrical and Electronic Engineers. IEEE Std. 1003.1c-1995 POSIX Part 1: System Application Program Interface-Amendment 2: Threads Extension, 1995.