Experimental Assessment of Scenario-Based Multithreading for Real-Time Object-Oriented Models: A Case Study with PBX Systems

Saehwa Kim[1], Michael Buettner[1], Mark Hermeling[2], Seongsoo Hong[1]

[1] School of Electrical Engineering and Computer Science,
Seoul National University, Seoul 10 52 80, South Korea
{ksaehwa, buettner, sshong}@redwood.snu.ac.kr
[2] IBM Software Group, IBM Singapore Pte Ltd, 9 Changi
Business Park Central 1, Singapore 486072, Singapore
hermilin@sg.ibm.com

Abstract. This paper presents an experimental evaluation of our scenario-based multithreading for real-time object-oriented models by the use of a case study of a Private Branch eXchange (PBX) system. The PBX system was taken from the industry and exhibits a number of characteristics found in real-world applications such as a highly reconfigurable dynamic structure and a typical layered architecture. The objective of this experimental study is to assess the improvements to 1) the modeling environment in terms of ease of use for designers and 2) the performance of the resultant executables. We show how our toolset was applied to the PBX system to model scenarios, as well as to generate a scenario-based multithreaded executable. The study clearly shows that our method can handle large-scale, complex models and that scenario-based multithreading achieves the performance improvements for a real-world model.

1. Introduction

Real-time embedded systems are becoming increasingly sophisticated and complex, while at the same time experiencing a shorter time-to-market with greater demands on reliability. As a result, the need for systematic software development methods and tools for real-time embedded systems is now greater than ever.

Recently, the Object Management Group (OMG) [17] initiated Model Driven Architecture (MDA) [16] as an approach to supporting model-to-code bridges. This clearly shows the high demand for the ability to generate executable applications directly from object-oriented models. MDA uses the upcoming revision of the Unified Modeling Language (UML) [22], UML 2.0, to allow modeling of executable architectures. Using this new revision of the industry standard modeling language, designers can raise the abstraction level and stop worrying about implementation level concepts like tasks and mutexes and instead focus on the desired behavior of their systems.

However, current modeling tools for object-oriented modeling, such as IBM Rational RoseRT [8], ARTiSAN Real-Time Studio [1], I-Logix Rhapsody [9], and IAR visualSTATE [7], lack in providing predictable and verifiable timing behavior and the automatically generated code is not always acceptable. For real-time embedded systems it is of the utmost importance to generate executables that can guarantee timing requirements with limited resources. Currently, designers must map design-level objects to implementation-level tasks in an ad-hoc manner. Because task derivation has a significant effect on real-time schedulability, tuning the system with this approach is often extremely tedious and time-consuming.

In our previous work [11, 12], we have proposed a systematic, schedulability-aware method of mapping object-oriented real-time models to multithreaded implementations. This is based on the notion of scenarios. A scenario is a sequence of actions that is triggered by an external input event, possibly leading to an output event [11]. In [12], we presented a multithreaded implementation architecture based on mapping scenarios to threads. This is contrary to the architecture found in current modeling tools that map a group of objects to a thread. In [14], we presented a complete tool set implementation of the scenario-based multithreading architecture for UML models as well as experimental results that validate this implementation. Our implementation exploits an established UML modeling tool, RoseRT [8], by designing a scenario-based run-time system that maintains backwards compatibility with the RoseRT run-time system.

In this paper, we present an experimental evaluation of our scenario-based multithreading of real-time object-oriented models. The objective of this experimental study is to assess the improvements to the modeling environment in terms of ease of use for designers and performance of the resultant executables. For this study, we have chosen a Private Branch eXchange (PBX) system as our target embedded real-time system. To show the benefits of our approach for a real world model, we acquired the model from an industry source. The PBX system model we adopted for the case study exhibits a number of characteristics found in real-world applications such as a highly reconfigurable dynamic structure and a typical layered architecture.

We show how our tool simplifies modeling by achieving a distinct separation between design and implementation with respect to multithreading, while providing a method of modeling scenarios that is essentially associated with user-perceptible timing constraints. We also present experimental results that clearly demonstrate the performance improvements that can be gained by the scenario-based implementation generated by our tools.

1.1. Related Work

There have been several research efforts that have focused on the automated implementation of real-time object-oriented designs and associated schedulability analyses [4, 18, 20]. However, these approaches are applicable to a system design only after tasks have been completely identified, and do not address schedulability-aware mapping of real-time object-oriented models to implementations. Thus, real-time designers still need rigorous methods to efficiently achieve such mappings.

In [19], Saksena et al. addressed problems associated with automated code synthesis from real-time object-oriented models. As in our approach, they attempted to maintain a separation of design and implementation models. Though they presented a seminal approach for the automated implementation of real-time object-oriented designs, it was not comprehensive as they presented only guidelines and heuristics. Their approach is also different from ours in that they do not support scenario-based multithreading.

As UML has become the de-facto industry standard for software modeling, several research efforts have developed methods to design real-time embedded systems using UML [2, 3, 5]. These efforts are limited to exploiting UML at the design stage and fail to give solutions for generating code with desired timing behavior. There has also been research activities focused on model transformation in the UML framework [6, 15] that provide various model transformation techniques where transformations are specified in UML. These techniques can be integrated with our approach to derive intermediate models of scenarios and logical/physical threads.

The remainder of the paper is organized as follows. Sect. 2 summarizes UML 2.0 that we chose as our real-time object-oriented language and presents an overview of our scenario-based multithreading, comparing it with traditional structured-class-based multithreading. Sect. 3 describes the PBX system that we used as an example case study system. Sect. 4 explains how our toolset was applied to the model to generate a scenario-based multithreaded executable. Sect. 5 presents the results of our experimentation, comparing the performance of a structured-class-based implementation and our scenario-based implementation for the PBX model. The final section concludes the paper.

2. UML 2.0 and Scenario-Based Multithreading

In this section, we provide an overview of UML 2.0, our chosen real-time object-oriented modeling language and our scenario-based multithreading.

2.1 UML 2.0 Modeling Language

UML 2.0 is a general purpose modeling language developed by the OMG, and contains corrections and new content based on user feedback on the UML 1.x modeling language. It has been developed to properly represent complex, event-driven, potentially distributed real-time and embedded systems.

The basic element of model construction in UML 2.0 is a structured class. A structured class represents an object within the system that communicates with other structured classes exclusively through interfaces called ports. A finite state machine, represented by a state diagram, represents the behavior of a structured class. Receiving messages via ports causes the state machine to make transitions, executing the logic contained in the structured class.

The full behavior of a system is defined by the composition of all structured classes, their connections, and their state machines. The structure of a structured class is defined in a structure diagram. In this diagram other classes can be used as parts of the composition. These are referred to as structured-class-parts. A structured-class-part can be fixed, optional or plug-in. All fixed structured-class-parts contained in a system are instantiated when the system is initialized. Alternatively, a structured-class-part can be marked as optional or plug-in and such a structured-class-part is instantiated dynamically according to the needs of designers. They are not instantiated at initialization but must be explicitly created and destroyed by a state transition. A plug-in structured-class-part is not an actual instance, but is a reference to an existing structured class instance in the model, and is created by importing a reference to an instance of an incarnated optional or a fixed structured-class-part.

Another concept in UML is replication of structured-class-parts and ports. Each individual instance of a replicated structured-class-part can be accessed by using the replication index. In Fig. 1, the `PhoneProxy` is a replicated structured-class-part; there are multiple instances of `PhoneProxy` in `ProxyManager`, but it is modeled as one structured-class-part. Replicated ports can be understood in much the same way. A structured class may require multiple instances of one port and so the port is replicated. Messages may be sent from all of the port instances at once or they may be sent from one particular instance by specifying the port index. In Fig. 1, we can see that the port connecting the `ProxyManager` and `OAMSubsystem` structured-class-parts is replicated so that each instance of `PhoneProxy` has a discrete connection to `OAMSubsystem`.

For our toolset, we exploited IBM Rational Software Rose RealTime (RoseRT), which is a modeling tool that allows users to design object-oriented real-time systems using UML 2.0 and generate complete executables directly from these designs.

2.2 Scenario-Based Multithreading

In structured-class-based multithreading the entity which can be manipulated is a message. It is possible to map the incoming messages of a structured class to a certain thread, and possible to map a single message to a thread or assign it a priority. But in most cases the designer does not conceptualize in terms of individual messages, but in terms of message chains. It is more natural that an entire message chain would be mapped to a thread, or timing metrics would be considered from the start of a chain to the end.

Also, it is not possible in structured-class-based multithreading for a message coming into a structured class to be processed on different threads in different situations. This imposes great limitations on the designer. Our scenario based multithreading allows the user to define priority and thread mapping for a complete message chain instead of individual messages. Structured classes will execute on different threads at different times depending on which scenario message sequence it is participating in at the moment. This not only is more akin to the way a designer would conceptualize a problem, but it also allows much greater flexibility in model design.

Moreover, structured-class-based multithreading may degrade the performance of real-time systems by extending blocking time unnecessarily. The sources of blocking in structured-class-based multithreading are 1) two-level scheduling, 2) message sending, and 3) run-to-completion semantics as addressed in [18]. Blocking due to two-level scheduling occurs when a message is handled by a lower priority thread. Blocking due to inter-thread message passing occurs because the per-thread message queue is accessed by multiple threads. Finally, blocking caused by run-to-completion semantics is due to the synchronization requirements of each state transition of a structured class. This last type of blocking can occur for each instance of inter-thread message passing.

Blocking due to two-level scheduling can be eliminated if thread priorities are dynamically changed according to the priorities of the handled messages, and blocking due to message passing can be bounded as once for each task if IIP (Immediate Priority Inheritance Protocol) [10, 13] is adopted. However, blocking due to run-to-completion semantics can be neither eliminated nor bounded as once in structured-class-based multithreading. Consequently, scenario-based multithreading performs better than structured-class-based multithreading since it 1) eliminates the blocking due to inter-thread message passing that cannot be avoided in structured-class-based multithreading and 2) bounds as once the blocking due to run-to-completion semantics that may occur whenever messages are delivered between threads. In scenario-based multithreading, priority inversion has an upper bound of the duration of the processing of a single message by the scenario causing blocking. A more in depth discussion of our scenario-based multithreading approach can be found in [12].

3. PBX System: An Example Case Study System

As a case study, we made use of a Private Branch eXchange (PBX) phone system for servicing cell phones. We were fortunate to have the chance to perform our case study on a model acquired from an industry source. One of the uses of a PBX is to allow in house calling for an office or building without the need to use outside lines. This is achieved by mapping a telephone number to an extension, which is a physical device or jack. When a user picks up a phone connected to one of these extensions and dials a number, the PBX system identifies which extension is associated with the dialed number and connects the two extensions.

Our PBX system model exploits 29 structured classes and a high level of functionality. It is a typical layered model where the bottom hardware layer processes external inputs. The model consists of four top level structured classes: *ProxyManager*, *DeviceManager*, *OAMSubsystem*, and *CallController*, as in Fig. 1.

The *ProxyManager* manages a group of interfaces, *PhoneProxies*, between the physical phone devices and the PBX, while the *DeviceManager* maintains a group of representations of the physical phones. The *OAMSubsystem* is responsible for storing the mappings between telephone numbers and extensions, and is the mechanism used to check if a phone number is valid. The *CallController* maintains representations of calls that are currently active in the system, and these act as communication channels between *Phone* instances when a call is in progress.

A call is established as follows. When a phone powers on, the power on signal is received by the *ProxyManager* associated with the phone and is forwarded to the *DeviceManager* which creates a *Phone* instance which will act as the internal representation of the powered on phone. When the phone dials a number, the digits are buffered by the *PhoneProxy* until the send signal recalls the complete dialed number and sends it to the associated *Phone* instance, which then makes use of the *OAMSubsystem* to check if the dialed number is valid. If it is, the *Phone* instance will send a message to *CallController* requesting a new *Call* instance to be created. The created *Call* instance will then contact the *Phone* instance which represents the dialed phone. If the dialed *Phone* instance is not busy, the *Call* instance will cause the dialed phone to ring. If the dialed phone answers the call, a communication channel is established between the caller and the called phone and notification of the connected call is sent to the two *PhoneProxy* instances.

If an error occurs at any of these steps, for example the number is invalid or the dialed phone is busy, appropriate messages are sent to the related *PhoneProxies*. When a phone sends the signal to end the call, the two *Phone* instances are returned to a waiting state and the *Call* instance that was mediating the call is destroyed. When a phone is powered off, the corresponding *Phone* instance is destroyed.

4. Application of Our Scenario-Based Tool Chain

Our scenario-based tool chain exploits an established UML 2.0 modeling tool, RoseRT, and is facilitated by 1) the RoseRT IDE where the PBX system model is integrated with our test harness, 2) our analyzer tool that derives a scenario model from code generated from RoseRT, 3) our code modifier that converts the single threaded source code into scenario-based multithreaded code, and 4) our customized scenario-based run-time system based on the original RoseRT run-time.

Our analyzer tool analyzes the model by parsing the generated source code to derive a new model of the application. It detects scenario starting points and recognizes each replication of a port as separate, which allows us to map signals from

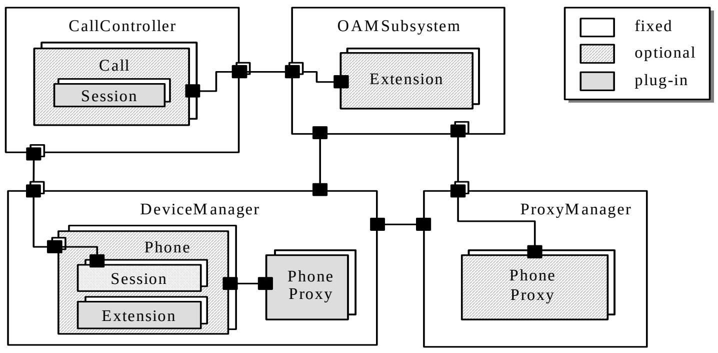


Fig. 1. Simplified structure diagram of our PBX system

different replication indices to different threads. We used this capability to model scenarios from lower indexed phones as having lower priorities than scenarios from higher indexed phones. The generated model represents the system as scenarios in a tree structure that depicts the possible executions or actions of the scenarios. The designer must assign priorities to each of the scenarios in the scenario model. Assigning viable priorities is the responsibility of the designer. This process can be aided by profiling tools that calculate or estimate worst-case execution time and analyze schedulability [11].

After the scenario model is generated, our modifier tool adapts the application source code generated by RoseRT for scenario-based multithreading. This integration modifies the scenario starting points to exploit our runtime system and inserts code for thread construction and destruction. Also, each of the capsules is assigned a priority ceiling to ensure proper scheduling for the system.

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. Our customized version of the RoseRT run-time system support scenario-based thread execution with IIP as described in Sect. 3.1 [10, 13]. We used the RoseRT run-time system 2001.03.00 compiled with GCC 2.95.3. The target environment was Sun Solaris 9 (SunOS 5.9) on a Sun Microsystems Sun Blade 1000. The structured-class-based multithreaded implementation was adapted from the single-thread model and the mapping of structured classes to threads was done based on the guidelines described in [18].

5. Experimental Performance Results

In this section we report experimental performance results from our case study to show the performance improvements, compared to structured-class-based

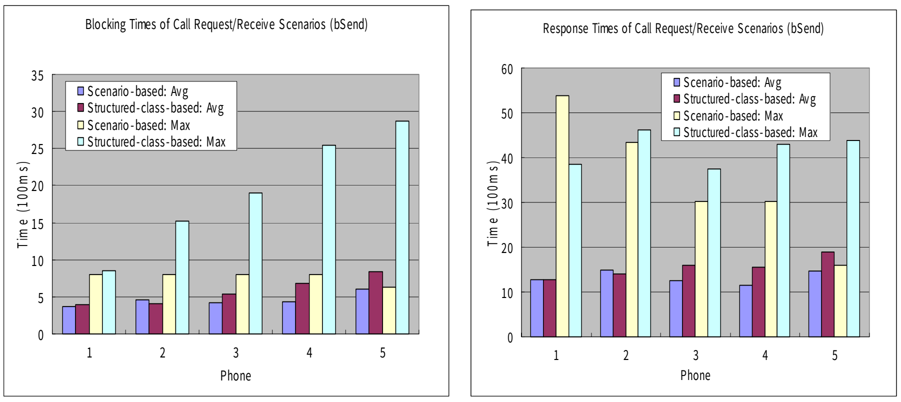


Fig. 2. (a) Blocking times and (b) response times for call request/receive scenario

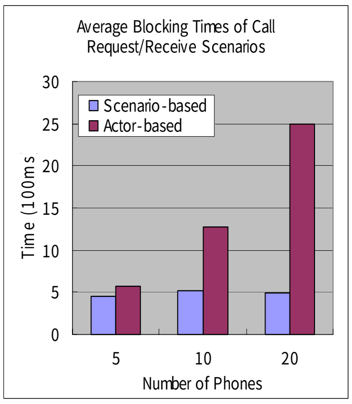
multithreading, that can be achieved with our scenario-based multithreading. Our results clearly show an improvement in performance with respect to blocking time and also scenario response times.

We performed experiments varying the number of phones from 5 to 100, measuring blocking times and response times for each scenario. Response time is the time from when the initiating external message is enqueued, until the last message in the execution chain is processed. Blocking time is the time that a scenario must wait for tasks to execute that have a lower or equal priority. We present the results for the call request/receive (bSnd) scenario and omit the results for other scenarios since they show similar results. Because the PBX system showed similar behavior with a various number of phones, we present blocking and response times for a system with five phones. We also show results for a varying number of phones to compare the scalability of the two implementations.

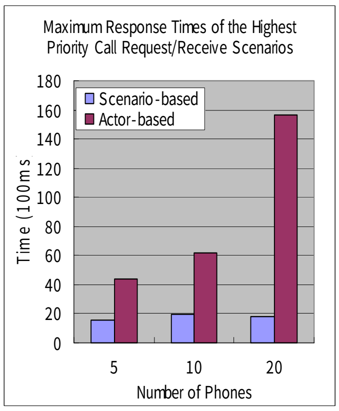
5.1. Blocking and Response Times

Fig. 2 shows the average and maximum blocking times (Fig. 2a) and response times (Fig. 2b) for the call request/receive scenarios. As shown in the figure, the blocking/response duration is generally shorter than for the structured-class-based implementation, especially considering maximum blocking/response times. The blocking/response time incurred by the structured class implementation increases with a higher priority, but this is simply due to the fact that for a high priority task, there are a greater number of tasks with a lower priority. Since the structured class approach processes messages in a first-in-first-out manner, a greater number of lower priority tasks create more blocking time. In scenario-based multithreading, a high priority task will always execute before a low priority task, so the blocking times do not significantly increase as priority increases.

Fig. 2b shows that the maximum response times for the structured class version are



(a)



(b)

Fig. 3. (a) Average blocking times and (b) maximum response times of the highest priority scenarios fairly consistent for all priorities, with some variation. On the other hand, for the scenario based-implementation the maximum response times drop consistently as priority increases. These results show that response times for the scenario-based implementation are nearly always lower than for the structured class implementation, and higher priority tasks benefit enormously from our scenario-based approach.

5.2. Scalability

To compare the scalability of the two multithreading approaches, we show the average blocking times (Fig. 3a) and maximum response times (Fig. 3b) results for `bSnd` scenarios for a varying number of phones. Other results such as maximum blocking times and average response times are omitted because they vary as would be expected from the results of Sect. 5.1.

As shown in Fig. 3a and 3b, the times for scenario-based multithreading are nearly constant as the number of phones increases, while those for structured-class-based multithreading increase dramatically. These results clearly show that scenario-based multithreading scales far better than the structured-class-based approach.

6. Conclusion

We have presented a case study to experimentally evaluate our scenario-based multithreading of UML 2.0 models. For this we used a PBX model from an industry source as a real-world example. We first described our UML PBX system model focusing on its structural and behavioral design. Then, we showed how our toolset was applied to the model to generate scenarios, as well as to generate a scenario-based multithreaded executable.

This study clearly showed that our method can handle large-scale, complex models and that scenario-based multithreading achieves the performance improvements in a real-world model. The study also showed the improvements to the modeling environment in terms of ease of use for designers, as we were able to quickly generate executables with the desired behavior without modifying the original model.

The performance results clearly showed a significant improvement in response times and a reduction in blocking times with scenario-based multithreading. We also noted that performance improvements over the structured-class-based architecture are more prominent in large-scale systems with a larger number of threads. These results show that our scenario-based multithreading is not only viable as a means to eliminate the manual thread assignment required in structured-class-based architectures, but also provides significant performance gains.

In the future, we will continue our research based on other real-world applications including support for distributed systems. We are also considering the potential application of quality of service concepts or models to our research.

References

1. ARTiSAN Software Tools Incorporation. Real-Time Studio, http://www.artisansw.com
2. B. P. Douglass. Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns, Addison-Wesley, 1999.
3. B. P. Douglass. Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley, 1999.
4. D. Gaudrean and P. Freedman. Temporal analysis and object-oriented real-time software development: A case study with ROOM/objectime. In Proceedings of IEEE Real-Time Systems Symposium, 1996.
5. H. Gomaa. Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley Longman, 2000.
6. W. Ho, J. Jézéquel, A. Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Proceedings of Automated Software Engineering (ASE'99), 1999.
7. IAR Systems Incorporation, visualSTATE, www.iar.com
8. IBM Rational Software Corporation. Rational Rose RealTime User Guide: Revision 2001.03.00, 2000.
9. I-Logix Incorporation. Rhapsody tools. http://www.ilogix.com
10. Institute for Electrical and Electronic Engineers. IEEE Std. 1003.1c-1995 POSIX Part 1: System Application Program Interface-Amendment 2: Threads Extension, 1995.
11. 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, 2000.
12. 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.
13. 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. 2002.
14. J. Masse, S. Kim, and S. Hong. Tool set implementation for scenario-based multithreading of UML-RT models and experimental validation. In Proceedings of IEEE Real-Time/Embedded Technology and Applications Symposium, 2003.
15. D. Milicev. Automatic model transformations using extended UML object diagrams in modeling environments. In IEEE Transaction on Software Engineering, vol. 28, no. 4, 2002.
16. J. Mukerji and J. Miller. Model Driven Architecture (MDA) Guide Version 1.0.1 OMG Document Number: omg/2003-06-01, 2003.
17. Object Management Group (OMG). http://www.omg.org.
18. 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, 1997.
19. M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-time object-oriented models. In Proceedings of IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2000.
20. M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz. Schedulability analysis for automated implementations of real-time object-oriented models. In Proceedings of IEEE Real-Time Systems Symposium, 1998.
21. B. Selic, G. Gullekson, and P. T. Ward. Real-time object-oriented modeling. John Wesley and Sons, 1994.
22. Unified Modeling Language (UML). http://www.uml.org