Analytical Assessment of Scenario-Based Multithreading for UML 2.0 Models: A Case Study with Soccer Robot Systems

Saehwa Kim
Department of Information and Communications Engineering
Hankuk University of Foreign Studies

This paper presents an analytical evaluation of our scenario-based multithreading for UML 2.0 models by the use of a case study of a soccer robot system. The objective of this analytical study is to assess the improvements to the response times of mission-critical scenarios. We discuss the difficulty encountered by designers when using current object-oriented modeling tools to achieve timing-guaranteed implementations. As the cause of such a difficulty we present drawbacks of actor-based multithreading, which is widely used by current modeling tools. We show performance benefits of scenario-based multithreading by analyzing blocking times of mission-critical scenarios with actor-based multithreading and scenario-based multithreading. The case study clearly shows that scenario-based multithreading achieves the performance improvements for UML 2.0 models.

1. Introduction

Object-oriented design technology has proliferated and been successfully applied to a wide range of domains. Consequently, there have been a number of efforts to apply object-oriented technologies to the development of real-time systems. This has resulted in the availability of a wide variety of commercial real-time object-oriented or UML 2.0 [7] structured-class modeling tools on the market today. For example, modeling tools such as IBM Rational RoseRT [5], ARTiSAN Real-Time Studio [3], IBM Telelogic Rhapsody [4], IBM Telelogic Tau [6] and IAR visualSTATE [8] render it possible for designers to model real-time systems, analyze them via executable models, and generate executable code for the systems.

However, current object-oriented modeling tools fail to provide predictable and verifiable timing behavior for the automatically generated executables. This lack of timing guarantees is largely due to the difficulty in identifying feasible task sets from object-oriented models. In object-oriented modeling, a real-time system is viewed as a collection of concurrent active

---

1 This work was supported by Hankuk University of Foreign Studies Research Fund of 2008.
information objects, referred to as actors, communicating with each other via messages. According to this notion, current object-oriented modeling tools have widely adopted an actor-based multithreading architecture [9, 13-17] for the generated implementations. In actor-based multithreading, an actor is the entity which can be considered a task, and so can be mapped to a thread. Since actors collectively compose a tightly coupled communication network, tasks in actor-based multithreading are highly dependent on each other. This task model makes it difficult to automate task identification while maximizing real-time schedulability since the high level of dependency creates an extremely complex optimization problem.

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. In [11], 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 [12], we presented a complete tool set implementation of the scenario-based multithreading architecture for UML 2.0 models as well as experimental results that validate this implementation. Our implementation exploits an established UML 2.0 modeling tool, RoseRT [5], by designing a scenario-based run-time system that maintains backwards compatibility with the RoseRT run-time system.

In this paper, we present an analytical evaluation of our scenario-based multithreading of UML 2.0 models. This paper presents an analytical evaluation of our scenario-based multithreading for UML 2.0 models by the use of a case study of a soccer robot system [1]. The objective of this analytical study is to assess the improvements to the response times of mission-critical scenarios. We discuss the difficulty encountered by designers when using current object-oriented modeling tools to achieve timing-guaranteed implementations. We also present drawbacks of actor-based multithreading. We show performance benefits of scenario-based multithreading by analyzing blocking times of mission-critical scenarios with actor-based multithreading and scenario-based multithreading. The study clearly shows that scenario-based multithreading achieves the performance improvements for UML 2.0 models.

The remainder of the paper is organized as follows. Section 2 describes the robot soccer system that we used as an example case study system. Section 3 explains how it is difficult for designers to use current object-oriented modeling tools to achieve timing-guaranteed implementations. Section 4 discusses the drawbacks of actor-based multithreading, which is widely used by current modeling tools. Section 5 presents the benefits of scenario-based
multithreading by comparing blocking times of mission-critical systems from scenario-based multithreading with those of actor-based multithreading. The final section concludes the paper.

2. Soccer Robot System

Robot soccer is a game where two teams of autonomous robots compete to score goals [1]. Each soccer robot contains a camera, two motors, a micro-controller, and a wireless communication module. The soccer robot can receive two commands, stop and start, via a wireless communication device. Once the robot receives a start command it searches for the ball using information from the camera, calculates an optimal path to drive the ball, and then controls its motors to drive itself through the path.

To model the robot system, we used six actors as shown in the structure diagram in Fig. 1. The Communication actor receives commands from a wireless receiver and sends messages to the RobotControl actor. The Vision actor reads image data from the camera, distinguishes visible objects, and send this information to the Location actor which then uses this information to calculate the locations of the ball, goals, and other robots. The RobotControl actor receives messages from the Communication and Location actors, and determines the appropriate action of the robot. When RobotControl determines that the robot must move, it computes the path information and passes this to the PathTracker actor, which computes the necessary speed and direction for the motors and sends this information to the Motor actor. Finally, the Motor actor controls the motors.

![Fig. 1. A soccer robot system.](image-url)
3. Difficulty of Feasible Task Set Identification

In most current object-oriented modeling tools, the default task mapping assigns all actors in the model to a single thread. It can easily be seen that this single thread approach produces a soccer robot that may not function correctly. Consider a message sent from the Motor actor to the PathFinder actor. Such a message cannot be processed until the currently executing transition completes its execution, as all transitions are executed non-preemptively in the single thread implementation. With this, if the timeout interval of the Motor actor is shorter than the execution time of any transition in the system, a Motor timeout message may not be processed before the Motor times out again. To avoid this problem, we must refactor the application to break down all long transitions into a number of shorter transitions. However, this is not always possible.

To overcome the limitations associated with single thread implementations, actor-based multithreading has been widely adopted by current object-oriented modeling tools including RoseRT. This approach groups actors into tasks and assigns each task to a thread. To use this technique, designers must determine how many threads are necessary and which actors should be grouped together and mapped to the same thread. Adopting the guidelines presented in [16], we can create an actor-based multithread implementation of the soccer robot as shown in Fig. 2.

![Fig. 2. Actor-based multithreading of the soccer robot example.](image-url)
which is an annotated structure diagram combined with the behavior diagrams of the structured classes. By creating tasks according to [16], we attempt to reduce overhead due to inter-thread message passing and context switching. After identifying tasks, we map each task to a thread and assign rate monotonic priorities to the threads. As shown in the figure, there are four threads, \( th_{\text{mot}} \), \( th_{\text{obj}} \), \( th_{\text{vis}} \), and \( th_{\text{com}} \), with a decreasing priority order. While this identification of tasks is reasonable, it may not be a feasible implementation. If the task set does not meet the necessary timing requirements, it is necessary for designers to fine tune the task set. This process, using current modeling tools, is troublesome.

4. Blocking Time Analysis of Actor-Based Multithreading

The sources of blocking in actor-based multithreading are 1) two-level scheduling, 2) message sending, and 3) run-to-completion semantics as addressed in [16]. 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 an actor. 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 immediate priority inheritance protocol (IIP) [2, 10] is adopted. However, blocking due to run-to-completion semantics can be neither eliminated nor bounded as once in actor-based multithreading. This can be seen by performing a schedulability analysis on the identified task set of the soccer robot using the response time analysis shown in [14].

As an example, we use the transition sequence triggered by the timeout event of the RobotControl actor that consists of the timeout transition of RobotControl, the reqLocation transition of Location, the location transition of RobotControl, and the setPath transition of PathTracker. The calculated response time of this transition chain is as follows.
\[ R_{\text{sub}} = \left( \max(C_{\text{RobotControl}}) + \max(C_{\text{Vision}}) + \max(C_{\text{PathTrac}}) + \max(C_{\text{Motor}}) \right) \]
\[ + \left( C_{\text{location}} + C_{\text{Movement}} + C_{\text{PathTrac}} \right) \]
\[ + \left[ \frac{R_{\text{sub}}}{T_{\text{wor}}} \right] \left( C_{\text{Motor}} + C_{\text{PathTrac}} + C_{\text{Motor}} \right) \]  

where \( C_{b}^{A} \) denotes the worst case execution time of transition \( B \) in actor \( A \) and the blocking due to inter-thread message passing was not considered as in [16]. The first term is the run-to-completion blocking time and the second is the execution time of the transition chain. The last is the interference time caused by higher priority transition chains. As shown in the first term of Equation (1), run-to-completion blocking may happen for every instance of inter-thread message passing. Due to this large blocking term, the maximum response time of this transition chain may be larger than the timeout interval. As opposed to this, scenario-based multithreading eliminates the blocking due to inter-thread message passing and bounds as once the blocking due to run-to-completion semantics.

5. Benefits of Scenario-Based Multithreading

The behavior model in UML 2.0 structured-class models is based on event processing. As such, in multithread implementation architectures each thread receives its associated messages and processes them one by one. In actor-based multithreading, the transitions in an end-to-end chain may be executed on multiple threads because the transition chain may flow through actors which are mapped to different threads. With scenario-based multithreading, a scenario is a sequence of transitions flowing through an end-to-end computation initiated by an external event, and the entire transition chain is mapped to the same thread. Also, multiple scenarios can be mapped to a single thread and the priority of the thread will change dynamically according to the priority of the currently executing scenario.
Fig. 3 shows how scenario-based multithreading can be applied to the soccer robot example. In this example, we used four threads $th_{mot}$, $th_{rob}$, $th_{vis}$, and $th_{com}$ with a decreasing priority order. In actor-based multithreading, all transitions contained in the same actor must execute on one thread. For example in Fig. 3, the $reqLocation$ and $visionData$ transitions of the $Location$ actor cannot execute on different threads. To the contrary, in scenario-based multithreading $reqLocation$ executes on the $th_{rob}$ thread and $visionData$ executes on the $th_{vis}$ thread according to the scenarios to which they belong, as shown in Fig. 3. Also, a single transition can execute on two different threads if it is part of two scenarios; for example the $setPath$ transition in $PathTracker$. To compare the response times that result from scenario-based multithreading with those of actor-based multithreading, we calculate the response time of the scenario triggered by the $timeout$ event of $RobotControl$, as we did in Section 4.

$$R_{rob} = \max(C_{RobotControl} + C_{Location} + C_{PathTracker})$$

$$+ \left( C_{RobotControl} + C_{Location} + C_{RobotControl} \right)$$

$$+ \left( \frac{R_{rob}}{T_{mov}} \right) \left( C_{Motor} + C_{PathTracker} \right)$$

(2)
Compared with Equation (1) in Section 4, the first term of Equation (2), the run-to-completion blocking time, has been significantly decreased in scenario-based multithreading. This is because scenario-based multithreading eliminates inter-thread message passing and thus run-to-completion blocking is bounded as at most once, and is limited to the duration of one critical section of a lower priority transition.

6. Conclusion

We have presented a case study to analytically evaluate our scenario-based multithreading of UML 2.0 structured-class models. For this we used a robot soccer model as a real-world example. We first described our UML robot soccer system model focusing on its structural and behavioral design. Then, we analyzed blocking times of mission-critical systems with traditional actor-based multithreading and our scenario-based multithreading.

This study clearly showed that scenario-based multithreading achieves the performance improvements in UML 2.0 model. This study also shows that our scenario-based multithreading is viable as a means to eliminate the manual thread assignment required in structured-class-based architectures.

In the future, we will continue our research based on various real-world applications including support for distributed systems. We are also considering the potential integration of real-time requirement modeling to our research.

References

Analytical Assessment of Scenario-Based Multithreading for UML 2.0 Models


요약: 본 논문에서는 축구 로봇 시스템에서의 사례 연구를 통해 UML 2.0을 위한 시나리오 기반 멀티쓰레딩을 분석적으로 평가한다. 이 분석 연구는 긴급한 임무의 시나리오의 응답 시간의 향상을 평가하는 것을 목적으로 한다. 본 논문에서는 설계자가 시간 제약을 만족시키도록 소프트웨어를 구현하는데 있어서 현재의 객체 지향 모델링 도구를 사용할 때 당면하는 어려움에 대하여 논한다. 아울러 이의 원인으로서 현재 모델링 도구로서 널리 사용되는 액터 기반 멀티 쓰레딩의 단점에 대하여 논한다. 본 논문에서는 액터 기반 멀티쓰레딩과 시나리오 기반 멀티쓰레딩의 긴급 임무 시나리오의 흐름 정 시간을 분석하여 비교함으로써 시나리오 기반 멀티쓰레딩의 성능상의 장점을 보인다. 이 사례 연구는 시나리오 기반 멀티 쓰레딩이 UML 2.0 모델에 대하여 성능상의 향상을 가져오는 것을 명백하게 보여준다.

---

2 본 논문은 2008년도 한국외국어대학교 학술연구비 지원에 의해서 연구되었음