Scenario-Based Implementation Architecture for Real-Time Object-Oriented Models*

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

Abstract

This paper presents a scenario-based implementation architecture supporting a method capable of automatically mapping real-time object-oriented models into multi-threaded implementations. To implement the synthesis tool supporting the method, we exploit existing CASE tools that support the object-based implementation architecture. Challenges in our approach are (1) how to embed our implementation model into generated design-model-dependent code and (2) how to implement the model-independent run-time-system library.

In our approach, to map each scenario to a thread, we make external messages starting scenarios delivered to their mapped physical thread. The main operation of the thread is (1) waiting for any external message to be delivered and (2) executing a while loop where all internal messages are sent and received. The state transition of an active object is guarded by an object-specific mutex to maintain the run-to-completion semantics. The priority of a thread is dynamically set according to the scheduling attributes of an external message for the thread to process.

1. Introduction

Due to continuously increasing demands, real-time embedded systems are getting extremely complicated. Thus, it has become inevitable for real-time embedded system developers to rely on systematic software design methodologies. Among a wide variety of design methodologies, an object-oriented technology has become dominant since 1990's. This is due to its prominent benefits such as encapsulation, inheritance, polymorphism, component-based coding, etc. These allow for easy software reuse and maintenance.

However, in object-oriented design methodologies, it is not very obvious how to translate a design model into tasks that collectively form the real executable implementation. Note that task derivation has a significant effect on the real-time schedulability of the resultant system. Existing object-oriented CASE tools force designers to map objects to tasks in an ad-hoc manner. This requires tedious manual tuning of design models and task mapping.

We have proposed a systematic, schedulability-aware method that maps real-time object-oriented models to multi-threaded implementations in an automated manner [1, 2]. The proposed method uses the notion of scenarios[1] and preemption thresholds [3]. We define a scenario as an end-to-end computation from an external input event and possibly to an output event. It can be described as a sequence of events triggered by an incoming external event. The proposed method is a three-step process: (1) deriving scenarios, (2) identifying logical threads, and (3) identifying physical threads. The physical threads are the final implementation-level tasks. The proposed method maps mutually exclusive scenarios into logical threads, and assigns each logical thread a priority and a possible maximum preemption threshold, guaranteeing the schedulability of the whole system. Then, the method groups logical threads into mutually non-preemptive groups [4], each of which is mapped into a physical thread. This can significantly reduce the number of threads.

In this paper, we present the scenario-based implementation architecture supporting our proposed mapping method. To implement the synthesis tool supporting our method, we exploit an existing CASE tool that supports the object-based implementation architecture. The existing CASE tool (1) generates the design-model-dependent code and (2) links it with the model-independent run-time system library to build an executable binary. Note that the design-model-dependent code does not contain any implementation-model-dependent information. The model is composed of (1) the

* The work reported in this paper is supported in part by MOST under the National Research Laboratory (NRL) grant 2000-N-NL-01-C-136 and by the Korea Science and Engineering Foundation (KOSEF) grant R01-1999-00206

[1] In our previous papers, we used the term transactions instead of scenarios.

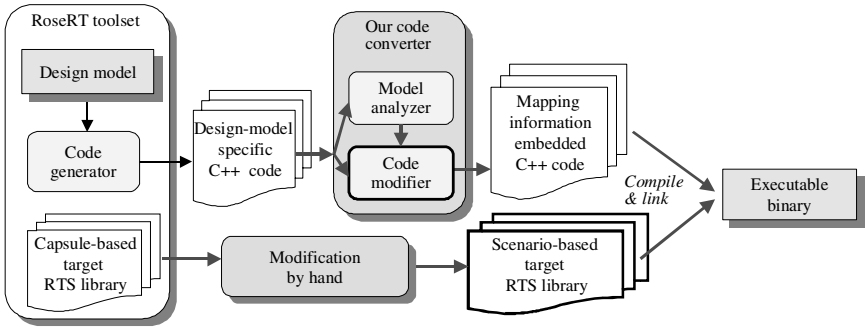

Figure 1. Supporting our proposed method using RoseRT.

design model and (2) implementation model. Programmers develop the design model and our method derives the implementation model for a given design model, which determines (1) how many threads are created and (2) to which thread each message is mapped.

With this, our challenges are (1) how to embed the implementation model into the generated design-model-dependent code and (2) how to modify the model-independent run-time-system library to make it fit to our scenario-based implementation architecture. The constraint is to guarantee that the resultant implementation shows the same operational behavior as that of the object-based mapping, including the run-to-completion semantics. Our solution approach is summarized as follows.

- To map each scenario to a thread, we make the first external messages starting scenarios delivered to the mapped thread.
- The main operation of the thread is (1) waiting for any external message to be delivered and (2) executing a while loop where all internal messages are sent and received.
- The state transition of an active object is guarded by the object-specific mutex for the run-to-completion semantics.
- The priority of a thread is dynamically set according to the scheduling attributes of an external message for the thread to process.

The remainder of the paper is organized as follows. In Section 2, we overview our implementation approach. In Sections 3 and 4, we describe specific problems and solutions for implementing the model dependent and independent parts, respectively. Finally, we conclude the paper in Section 5.

2. Implementation approach

We use UML-RT [5] as our source programming language as in our previous work. We also use UML-RT terminologies such as *capsule*, *capsule instance* and *capsule role*. They respectively represent the template (class) for an active object, the instantiated active object, and the reference to the active object. To begin with, we describe the requirements of our target platform. Then, we present our implementation approach, which exploits an existing CASE tool.

2.1. Assumption of the target platform

Our implementation requires that the scheduler of its target platform support following functionalities.

- Dynamic configuration of priority
- Reasonable range of priorities.
- The immediate priority inheritance protocol.

We assume that our implementations are targeted to such platforms.

2.2. Exploiting an existing CASE tool

To implement our mapping method, we exploit RoseRT [6], which is a CASE tool supporting UML-RT. Figure 1 shows our solution approach for this purpose. Programmers develop their design models with the RoseRT toolset. Then, they can generate the design-model-dependent C++ source code via the RoseRT code generator. Unless the programmers set some implementation specific configuration explicitly within the toolset, the generated code does not contain any implementation-model dependent data. More precisely, the default implementation model of the generated code is a single threaded process where all messages are mapped to a single thread.

With this generated code as input, our code converter produces C++ code conforming to our method. Our code converter is composed of (1) a model analyzer and (2) a

```
class LogicalThread {
  Controller*  targetThread;
  int          priority;
  int          preemptionThreashold;
}
```

(a)

```
class Capsule {
  Mutex      mtxCapsule;
  StructDef  structureDef;
  void       fsmBehavior();
}
```

(b)

Figure 2. Pseudo code for (a) the logical thread and (b) the capsule.

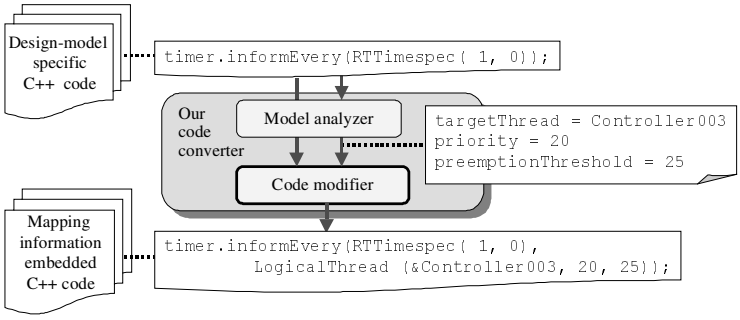

Figure 3. An example for the code conversion.

code modifier. A model analyzer resolves the model and finds a mapping according to our mapping method. Using the results of the model analyzer, the code modifier alters the generated code and adds implementation specific code to it. The concrete modification will be explained in Section 3.

The RoseRT toolset supports the target RTS (Run-Time System) library, which forms the model-independent implementation component. The target RTS library is linked with the generated code to build the executable binary. RoseRT provides the full source code of the RTS library so that programmers can modify and fit it to their specific target platform. We modify this target RTS library to make it conform to our scenario-based mapping strategy. We will describe this modification concretely in Section 4. We compile and link the converted generated code and modified target RTS to generate the executable binary.

With this implementation approach, a problem at hand is how to implement the bold-outlined diagrams in Figure 1, which are (1) the code modifier and (2) the scenario-based target RTS library. We explain the detailed problems and solutions in the following sections.

3. Implementing the model dependent code

In this section, we describe how to implement the code modifier that alters the design-model-specific code to reflect our implementation model. Note that its input implementation model is the output of our model analyzer. The code modifier directly determines the implementation architecture of the model dependent code. Specific problems for implementing the code modifier are summarized as follows.

- How to embed our scenario-to-thread mapping data into the design-model-specific code.
  - Where to embed the data structure representing the scenario-to-thread mapping data.
  - How to design the data structure representing the scenario-to-thread mapping data.
- How to modify the model dependent code to guarantee the run-to-completion semantics.
- How to embed the code for creating threads the method derives.

We explain the solutions for each in the following subsections.

3.1. Embedding the scenario-to-thread mapping data into the design model specific code

Where to embed the data structure representing the scenario-to-thread mapping data: Our solution is to use additional arguments of functions that register external messages to occur. The naïve solution is to embed it into all messages that can be sent. However, our mapping method groups all scenarios sharing an external input message into the same logical thread, thus to the same physical thread. Since implementation-level scenarios can be safely characterized with their external input messages, we embed scenario-to-thread mapping data only into external messages. Specifically, our code modifier embeds the mapping data into the functions that register external messages – such as timeout – to occur as their additional arguments.

How to design the data structure representing the scenario-to-thread mapping data: Our solution is to keep the attributes of each logical thread in the data structure. This is because scheduling attributes such as a priority and preemption threshold is assigned to a logical thread, and the implementation-level physical thread is just a group of logical threads. We define data structure `LogicalThread` for a logical thread as in Figure 2 (a), where `Controller` is the class type for a physical thread.

With these solutions, our code modifier adds argument LogicalThread, whose value is obtained from the output of the model analyzer, to each function that registers any external message to occur. Of course, our run-time

```
class Controller {
  MsgQueue     externalMsgQ;
  MsgQueue     internalMsgQ;
  pthread_t*   threadId;
  void         mainloop();
}
```

(a)

```
class Message {
  Message *    next;

  CapsuleRole  toCapsuleRole;
  Port         fromPort;
  short        signal;

  /* only for external messages */
  int          priority;
  int          preemptionThreadhold;

  void*        data;
}
```

Figure 4. Pseudo code for (a) the controller and (b) the message.

(a)

```
1 Controller::mainloop(){
2   while(1) {
3     waitForExternalEvents();
4     while(externalMsgQ is not empty){
5       foreach(extMsg with the highest priority) {
6         dispatch(extMsg);
7         while(internalMsgQ is not empty){
8           foreach(Message intMsg)
9           dispatch(intMsg);
10        }
11        threadSetPrio(max(priorities of
            external messages in externalMsgQ));
12      }
13    }
14  }
15}
```

(b)

```
1 Controller::dispatch (Message msg){
    /* if msg is an external message */
2   if (msg.priority is not null)
3     threadSetPrio(extMsg.preemptionThreshold);
4   msg.toCapsuleRole->mtxCapsule.enter();
5   msg.toCapsuleRole->fsmBehavior(
                msg.signal, msg.fromPort);
6   msg.toCapsuleRole->mtxCapsule.leave();
7   free msg;
8 }
```

Figure 5. Pseudo code for the thread's (a) main operation and (b) message dispatching.

system library provides these overloaded functions with an additional argument, as will be explained in the Section 4. Figure 3 shows an example code conversion that our code converter does. The model analyzer scrutinizes the design model from the RoseRT-generated C++ code and derives scenario-to-thread mapping data for each external message. The selected code in this example registers a 1 sec periodic timer. The code modifier adds the LogicalThread argument, whose value is initialized with the output from the model analyzer, to the corresponding function.

3.2. Modifying the design model to guarantee the run-to-completion semantics

To maintain the run-to-completion semantics of the real-time object-oriented model, state transitions in a capsule instance should be synchronized. In the object-based thread-mapping implementation, this semantics is naturally maintained because all state transitions in an object always occur within one thread. However, in our scenario-based mapping implementation, state transitions in an object may occur in more than one thread.

Using capsule specific mutex: Figure 2 (b) shows pseudo code for each capsule. The RoseRT code generator forms a class for each capsule in the given design model. As shown, we use a capsule specific mutex to synchronize state transitions of capsule instances.

Adoption of immediate priority inheritance protocol: To reasonably bound the run-to-completion blocking time, we adopt the IIP, or immediate priority inheritance protocol [7]. The reason that we adopt this protocol is because all threads always try to lock mutexes whenever they execute. The adoption of IIP in the context of preemption threshold scheduling is beyond the scope of this paper, so we do not further discuss about the protocol itself. With this adoption, a thread executing a scenario may be blocked only once before it starts its execution, either by the IIP blocking or by preemption threshold blocking. Our code modifier adds a code section that initializes this mutex for each capsule instance.

3.3. Creating threads the method derives

Our code modifier synthesizes the body of `initUpdateThreads()` of the `RTMain` class so that it creates all the derived physical threads. The RoseRT run-time system calls this function once during initialization.

4. Implementing the model independent run-time system library

This section describes how the model independent target RTS library is modified to support our scenario-based mapping implementation model. The concrete problems for implementing the run-time system are as follows.

- How to implement the operation of threads

```
class TimerNode {
  TimerNode*   next;

  Message*     msg;
  RTTimespec   timeout;
  Controller*  targetThread;
}
```

Figure 6. Pseudo code for the timer node.

(a)

```
1  Timing::informIn(RTTimespec timeout, LogicalThread lt){
2     Message msg = lt.targetThread->newMsg(
          capsule role calling this function, timer port, timeout signal,
          lt.priority, lt.preemptionThreadhold);
3     TimerNode tnode = newTimerNode (&msg, timeout, lt.targetThread);
4     insert tnode to the timeout callout queue;
5  }
```

(b)

```
1  RTTimerActor::sendTimeouts(void) {
2     dequeue all expired timer nodes from timeout callout queue;
3     foreach (TimerNode tnode) {
4        enqueue tnode.msg to tnode.targetThread->externalMsgQ;
5        tnode.targetThread->threadSetPrio(
            max(current priority, tnode.msg->priority));
6        free tnode;
7     }
8  }
```

Figure 7. Pseudo code of functions for (a) the registration and (b) the delivery of external messages.

- How to implement the external message registration
- How to implement the external message delivery

We describe solutions for each of them in the following subsections.

4.1. Implementing the operation of threads

Figure 4 (a) shows the class type for the (physical) thread, whose name is `Controller`. As shown, each thread has its own message queues. The data structure of the message is shown in Figure 4 (b), where `toCapsuleRole` is its target capsule role, and `fromPort` and `signal` are used as the input for the finite-state-machine behavior of the target capsule role.

Figure 5 (a) shows the basic code structure of each thread. As shown, the main operation of the thread is (1) waiting for any external message to be delivered and (2) executing a while loop where all internal messages are sent and received. An iteration of the body within the while loop from line 5 to line 12 corresponds to the execution of a scenario. Each message to be dispatched in line 9 corresponds to a message composing the message sequence of a scenario. When a thread finishes processing an external message and thus a scenario, it changes dynamically its priority to the highest priority of pending messages in its external message queue (in line 11).

Figure 5 (b) shows pseudo code for the message-dispatching operation. If a dispatched message is an external message (in line 2), the priority of a thread is dynamically set to the preemption threshold of the logical thread, to which the scenario for the thread to process is mapped (in line 3). The `fsmBehavior()` of a capsule in line 5 describes the finite-state-machine behavior of its owning capsule. It executes the appropriate action and transits the state of its capsule according to its behavioral definition. As explained in Section 3.2, this state transition is guarded by the capsule-specific mutex to meet the run-to-completion semantics (in lines 4 and 6). Note that the allocated message is freed in line 7 because the data structure is no longer needed.

4.2. Implementing the registration and delivery of external messages

Implementing the external message registration: As mentioned in Section 3.1, our run-time system library provides overloaded functions with an additional argument `LogicalThread`, for the functions that register external messages to occur. In RoseRT RTS, such built-in functions are timer services `informIn()` and `informEvery()` which are respectively for one shot and periodic timer registration.

Figure 7 (a) shows pseudo code for our `informIn()` function. It first allocates a memory chunk for a message that will be delivered as an external message (in line 2). It is allocated from the message pool of the target thread, to which the external message will be delivered. The allocated message is initialized as the target capsule role that calls this `informIn()` function, and external-message-type dependent port and signal, which are, in this case, a timing service port and timeout signal. The message is also initialized as the priority and preemption threshold of the target logical thread.

After that, in line 3, a timer node is allocated and initialized. The timer node is the data structure for the entry of the timer callout queue, whose pseudo code is shown in Figure 6. As shown, we make it contain a reference to the target thread. The timer node also has a reference to the external message to be delivered. The allocated timer node `tnode` is initialized with the

pointer to the message built in line 2 and the input arguments of the `informIn()` function, which are timeout value and target thread.

Implementing the external message delivery: Figure 7 (b) shows the pseudo code for the `sendTimeouts()` function of `RTTimerActor`, which is called whenever any registered timeout expires. First, it dequeues all expired timer nodes from the timeout callout queue. Then, for each dequeued timer node, it (1) enqueues the corresponding external message to the target thread (in line 4) and (2) adjusts the priority of the target thread, as the maximum of the current priority of the target thread and the priority of the message being enqueued (in line 5). After that, the allocated timer node is freed (in line 6).

As such, the priority of a thread is determined by the scenario, which the thread deals with. Whenever a thread is assigned to start a new scenario (in line 4), its priority is dynamically set (in line 5), according to the logical thread to which the scenario is mapped.

5. Conclusions

We have presented the scenario-based implementation architecture for real-time object-oriented models. The proposed implementation architecture aimed at supporting our previously proposed software-synthesis method, which automatically maps a given design model to multi-threaded implementations in a schedulability-aware manner.

To implement the synthesis tool supporting our method, we exploited RoseRT, a CASE tool for UML-RT. We provided (1) the code converter that consists of the model analyzer and code modifier and (2) modified run-time system library that fits to our scenario-based implementation architecture. The code modifier (1) embeds the logical-thread data-structure into external-message-registering functions as their arguments, (2) initializes the capsule-specific mutexes, and (3) synthesizes the body of the thread-initialization function to create threads that our method derives. The modified run-time system library has following features: (1) while there is no thread-specific mutex, the message dispatching operation is guarded by the capsule specific mutex. (2) There is no inter-thread message passing except the delivery of external messages such as a timeout signal. (3) The priority of a thread is dynamically set according to the scheduling attributes of an external message for the thread to process.

The main contributions of the paper are three folds. First, we have proposed the scenario-based implementation architecture, which is different from the object-based one supported by the current CASE tools. Second, we have described how our previously proposed method can be implemented exploiting existing CASE tools. Finally, we have presented how the preemption threshold scheduling is integrated into the scenario-based implementation architecture for real-time object-oriented models.

We are currently developing some performance metrics and testable object-oriented design models so that we can provide some experimental results, which compare our approach with others.

References

[1] S. Kim, S. Cho, and S. Hong, "Schedulability-Aware Mapping of Real-Time Object-Oriented Models to Multi-Threaded Implementations," *Proceedings of IEEE Real-Time Computing Systems and Applications Symposium,* pages 7-14, 2000.

[2] S. Kim, S. Cho, and S. Hong, "Automatic Implementation of Real-Time Object-Oriented Models and Schedulability Issues," *Proceedings of IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, pages 149-153, 2001.

[3] Y. Wang and M. Saksena, "Scheduling Fixed Priority Tasks with Preemption Threshold," *Proceedings of IEEE Real-Time Computing Systems and Applications Symposium*, pages 328-335, 1999.

[4] M. Saksena and Y. Wang, "Scalable Real-Time System Design Using Preemption Thresholds," *Proceedings of IEEE Real-Time Systems Symposium,* pages 25-34, 2000.

[5] Object Management Group, *OMG Unified Modeling Language Specification Version 1.3*, 1999.

[6] Rational Software, http://www.rational.com.

[7] Institute for Electrical and Electronic Engineers, *IEEE Std. 1003.1c-1995 POSIX Part 1: System Application Program Interface–Amendment 2: Threads Extension*, 1995.