Development Tools and Run-time System Architecture in SISAtime

Saehwa Kim

Department Information and Communications Engineering
Hankuk University of Foreign Studies, Yongin-si, Gyeonggi-do, 449-791 Korea.
E-mail: ksaehwa@hufs.ac.kr

In this technical report, we explain Scenario-based Implementation Synthesis Architecture with Timing Guarantees (SISAtime) toolset implementation and the run-time system architecture that supports SISAtime multithreaded code synthesis mechanisms. To facilitate understanding, we first explain the run-time architecture of the code basically generated by RoseRT. After that, we explain additional and extended items for SISAtime.

1. Extending RoseRT Development Tool for SISAtime

In this section, we describe the architecture of the SISAtime toolset. As shown in Fig. 1, the SISAtime toolset is an extension of RoseRT [1], which is one of the most famous real-time object-orientation tools. Specifically, we added a scenario extractor, a task extractor, a thread extractor, and a code modifier to the backend of RoseRT.

Developers create their system model using UML in the frontend of RoseRT, and the system model is transformed into a thread model by the scenario extractor, task extractor, and thread extractor. During this process, the tasks that will execute the system model are derived, and their scheduling attributes are automatically calculated. At the same time, the system model is synthesized into multi-thread code by the RoseRT code generator. However, the code synthesized here is designed so that all activities in one active object are executed by one thread and thus the generated code does not conform to the SISAtime thread model. The code modifier changes the code synthesized by RoseRT so that it

---

![Fig. 1. SISAtime toolchain structure.](image-url)
is consistent with the SISAtime thread model. All intermediate information generated and shared by the tools is in the form of XML. We explain these four additional tools in detail in the following subsections.

1.1 Scenario Extractor

The scenario extractor first derives the scenarios from the system model designed with RoseRT UML tool, and then analyzes the attributes of each scenario. Both processes are done mechanically. The first process that derives the scenarios is the same as that proposed in SISA, for which a detailed algorithm is presented in [2]. The scenario attributes that must be analyzed by the SISAtime scenario extractor are of four kinds: period, deadline, the set of active objects accessed by the scenario, and the worst-case execution time. Among these four, the period and deadline are extracted from the real-time profiling information of the sequence diagram. We obtain the set of active objects accessed by a scenario by analyzing all the activity code comprising the scenario. Since our previous work, SISA, also requires this information, the detailed analysis method is described in [2]. This information is needed in order to apply the IIP protocol. Besides, we need this information for schedulability analysis when we calculate the synchronization blocking time of a scenario.

Finally, the worst-case execution time is obtained experimentally via profiling in SISAtime. The methods for obtaining the worst-case execution times have been investigated in [3], but this is out of the scope of this paper. The problem of those analytical approaches is that the resultant times are too pessimistic, since the results tend to be based on exceptional cases that are not in normal modes. We took a more practical approach where we gathered the execution time information via the actual execution of the real code in the target machine. The scenario worst-case execution time is calculated as a summation of the worst-case execution times of its component activities. For profiling, we executed the whole system multiple times, while logging the execution times of each activity. From the logging information, we selected the longest execution time of each activity and set it as the worst-case execution time of the activity. To measure the execution time more accurately, this approach can be modified to employ profiling tools such as those in [3].

1.2 Task Extractor

The task extractor performs the following two tasks. First, it maps scenarios into tasks by analyzing the output information from the scenario extractor. For this, it builds a mapping table that records each scenario and the set of messages that triggers the scenario. By referring to this mapping table, the scenario extractor merges scenarios that are related with the same message into a task. After that, it assigns a proper priority and preemption threshold to each task. These also utilize the schedulability analysis algorithms of SISAtime.

1.3 Thread Extractor

The thread extractor merges mutually non-preemptive tasks into a single thread. The final thread model generated by the thread extractor contains the following information.

- SPP’s where messages arrive and the kinds of messages that trigger the scenarios executed by each task
- Scheduling attributes (priority and preemption threshold) of each task
- Set of active objects accessed by each task
- Mapping information from tasks to threads
- Stack size required by each task
1.4 Code Modifier

The code modifier converts the code synthesized by the RoseRT code generator into the code conforming to the thread model created beforehand. The code modifier first parses the existing synthesized code and then analyzes the system initialization routine, the routines for message delivery to SAP ports as well as general ports, and the main loop routines for each thread. It modifies those analyzed code segmentations so that the resultant code conforms to not only the derived thread model but also the SISAtime run-time system architecture, which will be explained in the next section.

2 Run-time System Architecture

In this section, we explain the run-time system architecture of the code synthesized by the SISAtime toolset. To facilitate understanding, we first explain the run-time architecture of the code basically generated by RoseRT. After that, we explain additional and extended items for SISAtime.

2.1 Run-time System Architecture of RoseRT

Fig. 2 (a) shows the run-time system architecture of the code generated by RoseRT. The threads comprising a system are categorized into user- and system-level ones. User-level threads are threads that execute the state machines of active objects. System-level threads are threads that are awakened by hardware interrupts and deliver messages to the SPPs of active objects.

When a system-level thread sends a message to a specific active object, it must find a user-level thread to handle that message. For this, the active-object-to-thread mapping table is used. This table records mapping information about which thread is to execute each active object. After a user-level thread to execute the message is found, the message is stored in a per-thread message queue that is maintained by the found user-level thread. The user-level thread that has
received the message fetches the highest priority message from its per-thread message queue one by one. After that, the thread runs the state machine of its mapped active object by selecting an activity to handle the message and execute the activity. When a message needs to be delivered to another active object in the middle of execution of the activity, the active-object-to-thread mapping table is referred to again and the message is stored in the message queue of the target thread. We have discussed how this architecture can be optimized in [2].

2.2 Modified Run-time System Architecture for SISAtime

Fig. 2 (b) shows the run-time system architecture of the code generated by the SISAtime toolset. Compared to the architecture of RoseRT, we do not use an active-object-thread mapping table. Instead we use a task-to-thread mapping table, PTS, and LIP. We also employ separate external and internal message queues. Specifically, the following items are added or modified:

- A user-level thread has two message queues: external and internal. The former is used for storing messages generated by system-level threads and the latter is for messages generated by user-level threads.
- Each message has attributes of priority and preemption threshold. The values of these attributes are set when the message is generated by a system-level task according to the task-to-thread mapping table.
- When a system-level thread generates a message, the task-to-thread mapping table is used to determine which user-level thread will receive the message. The delivered messages will be stored in the external message queue of the user-level thread.

```plaintext
▷ destPort is an SPP.
1 SPP-SEND (destPort, signal, data)
    ▷ table is the task-to-thread mapping table.
    ▷ Get an entry from the mapping table.
2 task ← table.get(destPort, signal)
    ▷ Create a message.
3 msg ← (destPort, signal, data, task.priority, task.threshold)
    ▷ Get thread to handle the message.
4 thread ← task.thread
    ▷ Send the message to the external queue of the thread.
5 thread.externalQueue.enqueue(msg)
    ▷ Set priority of the thread.
6 if thread is idle
7    thread.priority ← task.priority
8 else
9    thread.priority ← max(thread.priority, task.priority)
10 end-if
```

Fig. 3. Procedure used by system-level threads to send messages to SPP.
When a user-level thread receives a message in its external queue, it sets its priority and preemption threshold to that of the received message.

When a user-level thread sends a message, the message is always delivered to itself. Specifically, the message is stored in the internal message queue of the same thread.

Only when there is no message in the internal message queue of a user-level task, will the user-level task fetch a new message from its external queue.

While an activity is being executed, active objects and thread contexts are protected using IIP.

Now we investigate how such items are implemented at the code level. We explain a series of five steps that occur when the system is initialized and a scenario is released and executed. For simplicity, we discuss this implementation using pseudo-code.

First, when the system is initialized, user-level threads that will be used in the future are created. The number of threads is obtained from the XML file generated by the thread extractor.

Second, when the system is initialized, mutexes for all shared resources are created and initialized. Shared resources encompass all active objects and all user-level threads in the system. In accordance with IIP, the ceiling value of each

```
1 MAIN-LOOP (thread)
2   forever
3    ▷ Get a message from the external queue.
4     msg ← thread.externalQueue.dequeue()
5     threshold ← msg.threshold
6    ▷ A scenario starts here.
7     forever
8      ▷ Find the active object that owns the port
9       destObj ← msg.destPort.owner
10     ▷ Protect shared resources using IIP
11      thread.priority ← max(destObj.ceiling, thread.ceiling, threshold)
12      RUN-FSM(destObj, msg)
13     ▷ Stop protecting shared resources using IIP
14      thread.priority ← threshold
15     if thread.internalQueue is empty
16         break
17     end-if
18     ▷ Get a next message from the internal queue.
19     msg ← thread.internalQueue.dequeue()
20   end-forever
21    ▷ The scenario ends here.
22   thread.priority ← maximum of priorities of messages in thread.externalQueue
23 end-forever
```

Fig. 4. Procedure used by user-level threads to execute scenarios.
mutex is initialized as the maximum of the priorities of all tasks accessing the mutex. The information regarding which tasks access which shared resource is in the XML file generated by the thread extractor.

Third, when a system-level thread generates a message, it performs the procedure **SPP-SEND** shown in Fig. 3. It is largely composed of two sub-steps. The first sub-step (lines 2-5) is to find a user-level thread to process the created message and send the message. The second sub-step (lines 6-10) is to adjust the priority of the received thread to ensure that the thread processes the message. For the first sub-step, we use the task-to-thread mapping table (line 2). The second sub-step involves making the user-level thread run with the priority of the newly released task.

Fourth, the use-level thread executes a scenario by executing the procedure **MAIN-LOOP** shown in Fig. 4. This procedure first dispatches a message from the external message queue (line 3). This message, which was generated by the procedure **SPP-SEND()**, triggers a new scenario. The procedure **MAIN-LOOP** finds an active object to handle the message using the information in the message (line 6) and executes the state machine of the active object by invoking the procedure **RUN-FSM()** (line 8). While the state machine is executed, any generated messages are stored in the internal message queue. After the execution of the state machine finishes, the scenario continues to be executed by dispatching messages in the internal message queue one by one. Such a process is continued until there is no message in the internal message queue (line 10-12). In this process, the priority of the user-level thread dynamically changes. This is for implementing preemption threshold scheduling and protecting shared resources, which are the contexts of active objects and threads as follows.

- **Supporting preemption threshold scheduling (PTS):**
  - In PTS, the priority of a currently executing thread should be set to the preemption threshold of the thread. Therefore, while a scenario is being executed (lines 5-14), the priority of its thread is set to a value not less than the preemption threshold of the scenario (lines 7 and 9).
  - After the scenario ends its execution, the priority of the thread is set to the maximum of the priority of the waiting scenarios (line 15).

- **Supporting the immediate priority inheritance protocol (IIP):**
  - In IIP, the priority of a thread should be set to a value equal to or greater than the ceiling of shared resources while the thread is being executed. Therefore, while a state machine of an active object is being executed, the priority of the thread is set to the maximum ceiling of the active object (**destObj.ceiling**) and the ceiling of the thread (line 7).
  - After ending the execution of the state machine, the priority of the thread is restored to the previous value (line 9).

Finally, we use the procedure **PORT-SEND** shown in Fig. 5 when user-level threads deliver messages while ex-
executing activities. Contrary to the procedure SPP-SEND(), the procedure PORT-SEND() inserts messages into the internal message queue of the current thread. That is, the thread that sends the message and the one that receives it are the same. Messages delivered in this manner are dispatched in line 13 of the procedure MAIN-LOOP().

ACKNOWLEDGMENT

This work was supported by the Hankuk University of Foreign Studies Research Fund of 2010.

REFERENCES