Scenario-Based Multitasking for Real-Time Object-Oriented Models

Saehwa Kim, Jiyong Park, and Seongsoo Hong
{ksaehwa, parkjy, sshong}@redwood.snu.ac.kr

Abstract

Contemporary embedded systems quite often employ extremely complicated software consisting of a number of interrelated components, and this has made object-oriented design methodologies widely used in practice. To implement an object-oriented model in given target hardware, it is imperative to derive a set of tasks from the designed objects. This process of determining tasks and the events they handle greatly influences the real-time performance of the resultant system including response times and real-time guarantees. However, the innate discrepancies between objects and tasks make this exceedingly difficult, and many developers are forced to find their task sets through trial and error. In this paper, we propose Scenario-based Implementation Synthesis Architecture (SISA), an architecture consisting of a method for deriving a task set from a given object-oriented model and the development tools and run-time system architecture to support the method. A system developed with SISA guarantees the optimal response time for each event while deriving the smallest possible number of tasks. We have fully implemented SISA by extending the RoseRT development tool and applied it to an existing industrial PBX (private branch exchange) system. The experimental results show that SISA outperforms the best known conventional techniques by reducing maximum response times an average of 30.3%.

Keywords: object-oriented real-time system design, embedded software development methodology, automated multitasking code synthesis, object-oriented modeling tools, unified modeling language (UML).

1. Introduction

The rapid convergence of a multitude of disparate technologies has led to modern embedded systems that offer a diverse variety of features. The sheer complexity of these systems has resulted in extremely complicated software consisting of a number of interrelated components. To handle this complexity, there have been many efforts to import object-oriented design methodologies such as ROOM [39], OCTOPUS [3], and COMET [14]. Furthermore, there has been a growth of commercial modeling tools to support these methodologies such as IBM Rational RoseRT [20], ARTiSAN Real-Time Studio [2], I-Logix Rhapsody [21], Telelogic Tau [41], and IAR visualSTATE [19]. Tools such as these are not only used for creating models and running simulations, but also for the automatic generation of executable code. These tools have become widely adopted since they help developers effectively manage and maintain complicated, constantly evolving real-time embedded software.

To implement an object-oriented model in given target hardware, it is imperative to derive a set of tasks from the designed objects. This process of determining tasks and the events they handle greatly influences the real-time performance of the resultant system including response times and real-time guarantees. However, the innate discrepancies between objects and tasks make this exceedingly difficult, and many developers are forced to find their task sets through trial and error. We propose a task set derivation method that guarantees optimal response times and introduce development tools and a run-time system architecture to support this. To begin with, we investigate existing task set derivation methods and discuss their limitations. Following that, we explain our approach.

1.1. Motivations

In object-oriented design methodologies for embedded systems, a system is designed as a network of stereotyped objects called *active objects*. Active objects are objects that communicate with one another or outside of the contained system only by sending and receiving messages. Each active object has an internal state machine. According to the type of received message and its current state, the state machine selects and executes one of its handler routines called *activities*, by which the received message is processed. When an activity is executed, new messages can be created and sent to other active objects. Developers write code that activities will execute using programming languages such as C, C++, Java, etc. Object-oriented models designed with such active objects and time-constrained message flows are referred to as *real-time object-oriented models*. In many real-time object-oriented models, timing constraints are specified in message sequence charts [14], [35], [37], [38].

Conventional methods derive task sets using an active object as their basic unit [3], [10], [11], [14], [39]. These methods basically

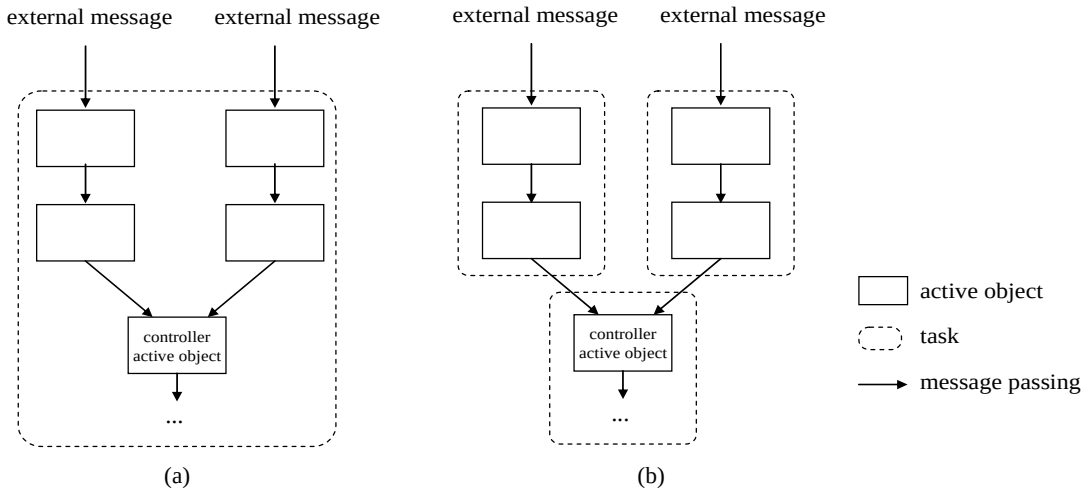

Figure 1. Example methods for making a task set when an active object is accessed by concurrent multiple message flows: (a) executes all active objects in a single task; (b) executes the shared active object in a separate task.

generate as many tasks as there are active objects in the system and let the same task execute all activities within one active object. Since too many tasks may be generated when task sets are derived this way, developers are also allowed to organize multiple active objects into groups and let a single task execute the activities for all the objects in the group. Considering these features, we call such a task set derivation method the *active-object-based task set derivation method*. This method has been widely used since it is intuitive and easy to be implemented. However, this method is known to be a non-optimal task set derivation method since it incurs unnecessary delays in system response times [13], [35], [38] in the following ways.

- Excessive blockings occur before the system can respond.
- Task contest switches occur excessively before the system can respond.

The overall process responsible for the occurrence of these delays is as follows. To simplify the implementation of the state machines, each task is bound by the constraint that it cannot process a new message until the currently executing activity terminates [13], [30], [35], [38]. Therefore, when a message is delivered between two tasks, the message may not be immediately processed and may be blocked if an activity is in the middle of being executed by the target task. Such inter-task message passing may occur multiple times before the system can respond to external events. As a result, systems experience a large number of blockings before they respond. Moreover, since blockings are accompanied by task context switches, systems experience an additional delay. Such delay factors are not due to the code of the activities written by developers but due to the fact that the task set derivation method is not optimized sufficiently.

Such time delay factors have undergone much analysis, and there has been a great deal of research aimed at reducing their occurrence. The most widely adopted approach is to let a single task execute all the activities in a group of active objects, thereby reducing inter-task message passing in the system. One method for grouping active objects proposed in [10], [11], [14] is for developers to identify messages that do not need to be processed concurrently and group the active objects that receive those messages. Another grouping method proposed in [35] is to pair two active objects where the maximum execution time of one object's activities is less than the maximum blocking time tolerable by the other object's receiving messages and vice-versa. However, such methods still have the following limitations.

- It is very difficult for developers to apply the proposed methods in practice.
- They are not optimal task set derivation methods since blockings and context switches may not be thoroughly eliminated.

To use the methods of [10], [11], [14], developers should analyze the patterns of how active objects send and receive messages according to their state machines. This is tedious work without the help of automated tool support. Likewise, it is very difficult work to determine the tolerable blocking times of all messages and to measure the worst-case execution times of all activities as required in [35]. Different external events may generate the same message, requiring the tolerable blocking time of that message to be adjusted at run-time according the importance of the current event. To fix this blocking time as one value for static analysis, we should use the worst-case value. This creates an unnecessarily strong constraint that tends to cause a waste of system resources.

Even if we could accept these limitations, the conventional task set derivation methods are open to further improvement since they

are not optimal methods. This is because the conventional methods can incur a large number of blockings and context switches for many systems in the following ways. Most embedded systems have active objects playing the role of the main controller as in Figure 1 [16], [48]. The figure shows that many of the message flows that should be processed concurrently pass by these active objects. As mentioned before, the conventional methods propose that all messages arriving at an active object should be processed in the same task. Therefore, the only way to eliminate inter-task message passing is to let one task process all messages arriving at every active objects as in Figure 1 (a). However, this creates a bottleneck where all messages that should be processed concurrently are processed in serial order in one task, further lowering the responsiveness of the system. Therefore, it is common to let a separate task process messages arriving at a controller object as in Figure 1 (b). This creates a large amount of inter-task message passing since most scenarios pass by the controller object. As such, using conventional methods, there are many cases where it is impossible to eliminate blocking and the accompanying context switches completely. Due to these problems, the actual state of affairs is that developers are using add-hoc methods where they make multiple task sets based on their experience and intuition, and then repetitively test whether the candidate task set meets desired response times.

On the other hand, most development tools support the ability to automatically generate executable code from the designed system models. In this process, developers describe their desired task set for the system to be executed with. We call this the *application of a task set*. However, the process of performing this as required by existing development tools is often troublesome and unreasonable. Due to these problems, developers have had a tendency to avoid using the code generation functions. Specifically, there are the following problems.

- To apply a task set, the object-oriented model describing the original behavior of the system should be modified.
- It is inconvenient for developers to apply a task set since standardized methods for this are not provided; even if provided, their functionalities are very restricted.

The task set application process should be accomplished separately from the object design process because the former is for representing the timing characteristics of the system while the latter is for modeling the original behavior. However, these two processes are not clearly separated in existing modeling tools, and thus there have been problems where developers have to modify object-oriented models whose design had been already finished. Such modifications are dangerous: the possibility for errors in the object-oriented models becomes especially great when developers repeatedly modify them to apply various task sets. Besides, most development tools do not support standardized methods for this process, or if they support some methods, their functionalities are very restricted. RoseRT is a typical example of a tool set that lacks support for this standardized method, requiring users to not only write code fragments for task creation and select active objects whose activities would be executed by specific tasks, but also to insert such code into object-oriented models manually. Rhapsody is an example of a tool that provides overly limited support: it provides a standardized method for grouping active objects, but subordinate active objects contained in different active objects cannot be grouped.

1.2. Solution Approach

We propose *Scenario-based Implementation Synthesis Architecture (SISA)* that solves these problems. SISA is an architecture consisting of a method for deriving a task set from an object-oriented system as well as the supporting development tools and run-time system architecture. The heart of SISA is a task set derivation method we call the *scenario-based task set derivation method*. This is a method that derives a task set that guarantees the optimal response time for each event with the smallest possible number of tasks.

This method has three important aspects. First, it introduces the notion of scenarios and lets all activities constituting a scenario be executed in the same task. Scenarios are a notion imported from [26], [29], [30], [36], where a scenario is a sequence of messages and activities that progress once triggered by an external event (message). This method makes inter-task message passing entirely unnecessary, and thus the associated blockings and context switches are completely eliminated. Second, it provides the notion of scenario groups and designates the scenario group as the unit of task creation. A scenario group is a set of scenarios triggered by the same external message. Each scenario group in the system is assigned its own task and all scenarios belonging to a scenario group are executed by the same task. This can reduce the number of tasks considerably compared to when a separate task is generated for each scenario. In addition to this, we can reduce the number of tasks further by letting multiple scenario groups execute in a task as needed. If developers designate scenario groups that do not need to execute concurrently, SISA executes them together in a single task, thereby further reducing context switching overheads and the memory size allocated to tasks. Third, it employs Immediate Priority Ceiling Inheritance Protocol (IIP) [6], [22], a real-time synchronization mechanism proposed in [6], to minimize the synchronization blocking that can be induced by the above mechanisms. The adoption of IIP enables systems to have optimal worst-case response times by

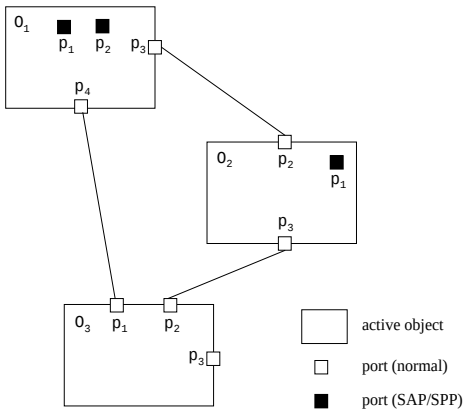

Figure 2. An example system modeled as a network of active objects.

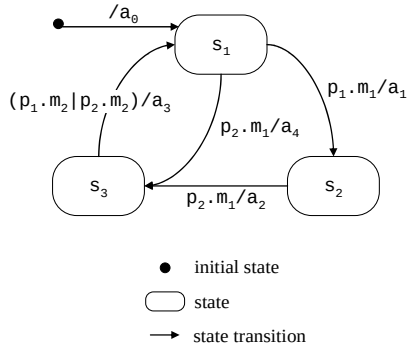

Figure 3. An example state machine that models the behavior of an active object.

guaranteeing that each scenario is blocked only one time before its code begins execution and not thereafter.

Besides the scenario-based task set derivation method, SISA also provides development tools and a run-time system architecture to support the method. First, development tools for SISA provide functions that analyze scenarios and scenario groups automatically and provide environments where developers can manipulate them. Developers can identify which scenarios exist in their systems without the need to analyze the state machines of active objects. Developers also can easily perform the work of assigning executing priorities to scenarios as well as binding scenario groups that do not need to be executed concurrently without modifying object-oriented models. Next, SISA provides a run-time system architecture that is different from conventional architectures. Conventional architectures use an *active-object-to-task* mapping table, by which the task that will process a message is determined by the active object that will receive the message. However, our architecture uses two mapping tables, which are the *scenario-group-to-task* table and the *scenario-to-priority* table. Using these tables, the task that will process a message is determined by the scenario group that the message belongs to, and the priority of the task is dynamically configured when a scenario branches off to a new scenario.

We have fully implemented SISA, extending the RoseRT development tool that uses UML 2.0 as a modeling language. We have also applied our SISA implementation to an existing industrial PBX system and evaluated its performance. This system was developed for industrial use and has many features found in complicated applications such as highly reconfigurable dynamic structures and typical layered architecture. The experimental results show that the maximum response times were reduced 30.3% on average and the response times for mission critical events were reduced by as much as 88.6%.

The rest of the paper is organized as follows. In the next section, we explain the system models that SISA assumes and make clear the problems that we aim to solve with SISA. In Section 3, we present the scenario-based task set derivation method, a key mechanism to solve these problems. In Section 4, we describe in detail the development tools and the run-time system architecture to support this method. In Section 5, we describe the performance evaluation results for systems developed with SISA. Section 6 describes related work and Section 7 concludes the paper.

2. Problem Description

In this section, we make clear the problems we aim to solve with SISA. To aid in understanding the rest of the discussion, we first explain the system models that we assume.

2.1. System Modeling

We model a system from two kinds of viewpoints. One is to model the system as a network of active objects. The other is to model the system as a set of scenarios. The former model represents what kinds of active objects reside in the system, how each active object behaves, how they are related with one another, where events external to the system are delivered, etc. We explain this model using the UML 2.0 structured class diagram [30] as a specific modeling language. The latter model represents what kinds of scenarios reside in the system, to which scenario group each scenario belongs to, which activities are executed in what order when each scenario is executed, etc. We devise a diagram called a *scenario tree* through which we can represent scenario groups and the scenarios contained

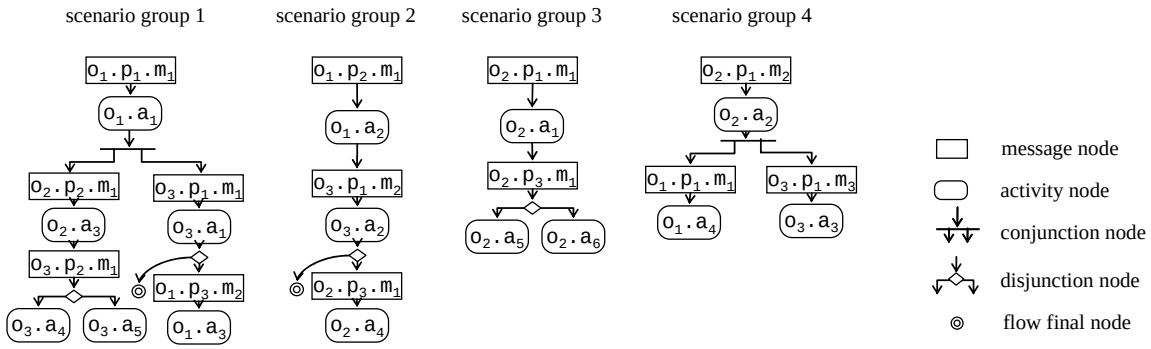

Figure 4. An example system modeled with scenario trees.

therein. Developers first model their systems as a network of active objects and then our SISA tool automatically translate the model into scenario trees. Now we explain these two kinds of modeling languages in order.

2.1.1. Modeling as a Network of Active Objects

In this modeling, a system is modeled as a network of active objects. Active objects are objects that communicate with one another or with outside of the contained system only by sending and receiving messages without method invocation. Each active object sends and receives messages asynchronously through interface objects called *ports*. Figure 2 shows an example system modeled in this way, where the large squares represent active objects and the smaller squares represent ports. If ports are connected, then it means that messages can be communicated through those ports. A message is composed of four items: an active object to receive the message, the message kind, data, and the message's priority. Among these, the message priority is used in deciding which message to process first when one or more messages are sent to an active object. A larger number represents a higher priority. We do not explain the other items since their meanings are apparent by themselves.

Generally, ports are used for passing messages between active objects. However, active objects in embedded systems should be able to send and receive messages with the external system environment which may include sensors, timers, motors, etc. For this, we adopt the notion of specialized ports in ROOM, namely Service Provision Points (SPP) and Service Access Points (SAP) [39]. The former is a port for receiving messages from external environments and the latter is a port for sending messages to external environments. These two kinds of ports are represented as small filled squares in Figure 2.

On the other hand, the behavior of an active object is modeled with a state machine. When an active object receives a message from its port, a state transition takes place according to the transition rules of its state machine. Which state transition to occur is determined by the kind of the received message. When a state transition occurs, its corresponding activity is executed, and the received message is processed. Developers write the code for each activity. The activity performs some computations based on data contained in the received message or sends messages to other active objects through ports. Figure 3 shows an example state machine for an active object. The label of a state transition, $p_i.m_j/a_k$ represents activity a_k being executed when a message of kind m_j arrives from port p_i.

In understanding the behavior of state machines, we should note that activities are executed in a non-preemptive manner according to their run-to-completion semantics [30]. These semantics are a kind of constraint condition that requires that any new activity be executed only after the activity currently being executed terminates. Such a constraint condition is used to allow multiple activities to access shared resources without synchronization and thus make it easy for developers to write code for activities [30].

2.1.2. Modeling as a Set of Scenarios

We also model a system as a set of scenarios. As mentioned previously, a scenario is a sequence of messages and activities that progresses once triggered by an external event (message). We define a set of scenarios triggered by the same external event as a scenario group. We devise and use *scenario trees* to represent scenario groups and the scenarios they contain. A scenario tree uses a tree format to represent how causally related activities progress once started by an external event while selectively branching off to multiple scenarios.

Scenario trees are extension of UML 2.0 activity diagrams and composed of five types of nodes and branches connecting them. The node types are 1) activity nodes which represent activities executing, 2) message nodes which represent messages being created and delivered, 3) conjunction nodes which represent scenarios dividing into two or more scenarios that execute in parallel, 4) disjunction

nodes which represent scenarios selectively branching off according to conditions determined at run time, and finally 5) flow final nodes which represent points where there is no further message generation. In a scenario tree formed this way, the tree represents a scenario group and each path from the root node to a specific activity node represents a scenario. Each scenario also can have a priority, and it is represented at the final activity node of the scenario.

Figure 4 shows the same system modeled with scenario trees that was modeled as a network of active objects in Figure 2. In this figure, the system is modeled with four scenario trees. The root node of each tree shows which external message initiates the corresponding scenario group. In scenario trees, a message is represented in the form $o_i.p_j.m_k$ and an activity in the form of $o_i.a_l$, where o_i, p_j, m_k, and a_l represent respectively the active object, port, message kind, and activity name.

2.2. Problem Statement

We want to solve the following problem for systems that can be modeled as above.

- Derive a task set that will execute a given system designed with active objects,
- such that the system has optimal response times, and
- the number of tasks in the task set is as small as possible,
- while designing the supporting development tools and run-time system architecture.

First, we define response time as the time elapsed from when the system receives a message from its external environment to when a specific activity terminates. That is, response time is not defined as one value for a system but defined for each scenario residing in a system. We call the amount of time each scenario takes to complete its execution the response time of the scenario. We also define a system as having optimal response time in the following way: when there are two or more concurrent scenarios, the most important scenario spends the least possible amount of time being blocked by less important scenarios. We call this optimal because delay factors are minimized for more mission critical scenarios. In addition to these conditions, the resultant task set should be composed of the smallest possible number of tasks. It is essential to reduce the number of tasks to decrease context switching overhead and ease the memory requirements for embedded systems. Finally, for this task set derivation method to be used in practice, development tools that allow developers to easily use this method should be provided. Also, a run-time system architecture that effectively supports this method should be designed.

3. Scenario-based Task Set Derivation Method in SISA

We propose *Scenario-based Implementation Synthesis Architecture (SISA)* that solves the aforementioned problems. The term SISA encompasses a method for deriving task sets, supporting development tools, and a run-time system architecture which together allow developers to produce optimal response times in a system designed with objects. In this section, we explain the scenario-based task set derivation method, the key notion of SISA. We discuss the development tools and the run-time system architecture in the next section.

The scenario-based task set derivation method is a method that derives a task set which minimizes both the response time for each scenario and the number of tasks in the system. This method incorporates three principal mechanisms. First, it lets all activities constituting a scenario be executed in the same task to eliminate inter-task message passing. Second, it prepares a task for each scenario group to reduce the number of tasks. It further reduces the number of tasks by allowing several scenario groups to be executed in a single task. Finally, it minimizes synchronization blocking using the Immediate Priority Ceiling Inheritance Protocol (IIP) [6]. We now explain these three mechanisms in detail.

3.1. Eliminating Inter-task Message Passing

SISA lets activities constituting a scenario be executed in the same task to eliminate inter-task message passing. In this case, a scenario proceeds along the following steps. 1) When the system receives an external message, a scenario is said to be released, and one of the tasks allocated for processing the scenario is selected and executed. 2) The task executes the state machine of the message's target active object by selecting and executing a specific activity. 3) The activity can generate and send messages to another active object in the middle of execution. 4) When the activity is terminated, the task executes step 2) again for active objects that have received messages. The process from step 2) to step 4) is repeated until no more messages are generated.

This fundamentally eliminates all inter-task message passing because the whole sequence of activities constituting a scenario is executed in one task. In addition to this, even those messages arriving at the same active object are processed in different tasks if they

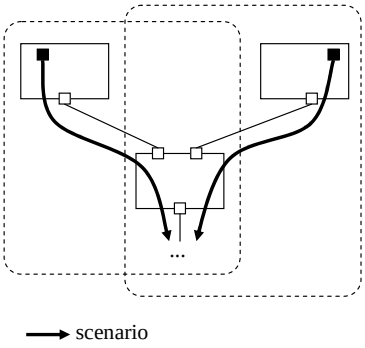

Figure 5. When multiple scenarios access the same active object, each scenario is executed in a different task. The legend is the same as Figure 1.

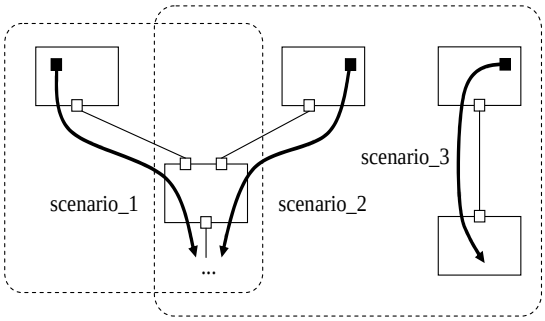

Figure 6. An example illustrating synchronization blocking. The legend is the same as Figure 1.

constitute different scenarios. Figure 5 illustrates this, where scenarios triggered by different external messages are executed in different tasks although there is an active object accessed in common by multiple scenarios.

3.2. Reducing the Number of Tasks

As explained above, all activities constituting a scenario are executed in a single task. This implies that a task is prepared for each scenario. However, a system usually has very large number of scenarios, and this generates too large a number of tasks. This in turn is accompanied by too much context switching overhead and memory consumption. To solve this problem, we reduce the number of tasks using the following two methods.

- A task is prepared for each scenario group instead of each scenario.
- If needed, we further reduce the number of tasks by binding multiple scenario groups and executing them in a single task.

To begin with, all scenarios in a scenario group start with the same priority. Therefore, even if the scenarios are executed in different tasks, they cannot preempt each other and are executed sequentially. That is, all scenarios are executed as if they were executed in a single task. As a result, the first method mentioned does not increase the response times of scenarios.

If developers ever need to further reduce the number of tasks, they can specify non-concurrent scenario groups to be executed in a single task. A typical example of this is when the messages triggering the release of two or more scenario groups have a clear, timed ordering relationship. For example, mobile robots receive sensor messages relaying the distances to certain obstacles and acknowledgement messages from the actuators when they have completed their motions. If the reception of sensor data and the generation of the first message are typically followed by actuator motion and the generation of the latter message, these two messages can be said to have a timed order relationship. Therefore, the scenario groups triggered by these two messages do not be executed concurrently, and we can execute them in a single task.

3.3. Minimizing Synchronization Blockings

If we use the methods explained so far, we can eliminate blockings due to inter-task message passing with the least possible number of tasks. However, synchronization blocking still occurs, meaning that blocking occurs when synchronization mechanisms are used to allow for different scenarios to access shared resources. Specifically, synchronization blocking can occur in the following two cases.

- If different scenarios share an active object, access to the shared active object is synchronized to maintain the run-to-completion semantics.
- If different scenario groups are executed in a single task, access to the context of the shared task is synchronized since multiple scenario groups share the task context.

Figure 6 illustrates such synchronization blocking with three scenarios, where scenario_2 and scenario_3 are mapped to the same task and scenario_1 to another task. The first synchronization blocking occurs between scenario_1 and scenario_2. As shown in the figure, these two scenarios can send messages to an active object concurrently. Since only a single activity can be executed within an active

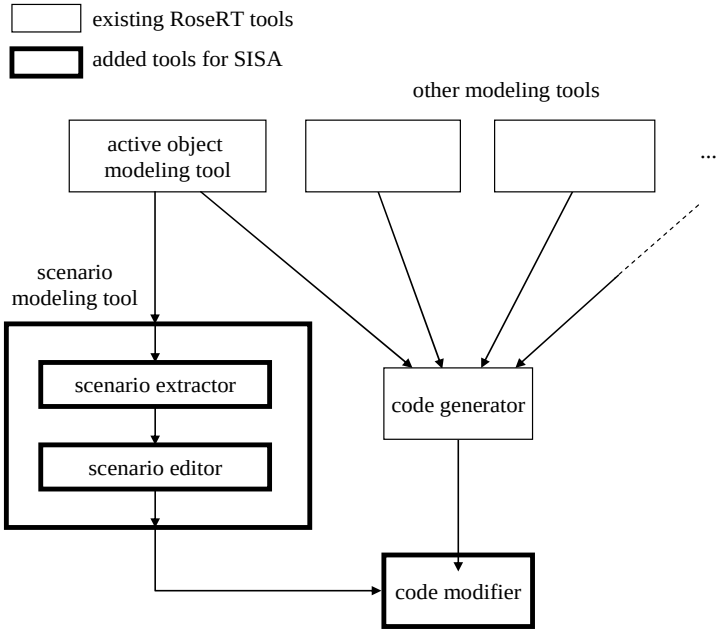

Figure 7. The configuration of the development tools supporting SISA.

```
1  -<scenarioBasedModel>
2     -<scenarioGroup id="1">
3        -<messageNode value="o1.p1.m1">
4           -<activityNode value="o1.a1">
5              <priority value = "2"/>
6              -<disjunctionNode>
7                 +<messageNode value="o2.p1.m2">
8                 -<messageNode value="o2.p1.m1">
9                    +<activityNode value="o2.a1">
10                    <priority value = "1"/>
11             </disjunctionNode>
12          </activityNode>
13       </messageNode>
14    </scenarioGroup>
15    +<scenarioGroup id="2">
16    +<scenarioGroup id="3">
   ......
17    -<MC>
18       <refScenarioGroup refID = "1"/>
19       <refScenarioGroup refID = "2"/>
20       <refScenarioGroup refID = "3"/>
21    </MC>
22    +<MC>
   ......
23 </scenarioBasedmodel>
```

Figure 8. An example xml file used by the scenario modeling tool.

object, if a scenario is in the middle of execution in this active object, then other scenarios cannot progress further and thus should be blocked. In this case the shared resources are active objects. The second synchronization blocking occurs between scenario_2 and scenario_3. In this case, there is no shared active object. However, if a task is already in the middle of executing an activity, another new activity cannot preempt the previous activity and occupy the task context, even if the new activity is contained in a different active object, since the shared task has one context. If this condition is not kept, the previous activity in the middle of execution may lose its context. Therefore, access to a task context should be synchronized and thus blockings occur.

To minimize these two kinds of synchronization blocking, we use Immediate Priority Ceiling Inheritance Protocol (IIP), a real-time synchronization mechanism proposed in [6]. IIP guarantees that every scheduling target is blocked only once before it begins to execute, and it is not blocked at all thereafter. Moreover, the blocking duration of a scheduling target is bounded as the longest critical section in the lower priority scheduling targets. We do not explain how IIP satisfies these conditions in detail since it is out of the scope of this paper. SISA treats scenarios as scheduling targets and applies IIP with active objects and task contexts as shared resources. As a result, all synchronization blocking of a scenario is completely eliminated except the first one-time blocking, and the duration of the blocking is bounded as the time spent by the longest activity in the lower priority scenarios.

4. Development Tool and Run-time System Architecture of SISA

In this section, we explain the design of the development tools and run-time system architecture that support SISA. We implemented the development tools by extending RoseRT [20], one of the most widely known development tools available. We also devised a run-time system architecture supporting SISA based on RoseRT.

4.1. Extending RoseRT Development Tool for SISA

Figure 7 shows the configuration of the development tools supporting SISA. The active object modeling tool and the code generator are tools supported by the existing RoseRT, and the scenario modeling tool and the code modifier are added tools to support SISA. First, the active object modeling tool is a tool that allows developers to model their systems as networks of active objects. Second, the code generator is a tool for generating executable code based on designed models using various modeling tools including the active object modeling tool. Third, the scenario modeling tool is composed of two subordinate tools: the scenario extractor and the scenario editor. The former analyzes the model designed with the active object modeling tool, extracts scenarios, and then models a system as a set of

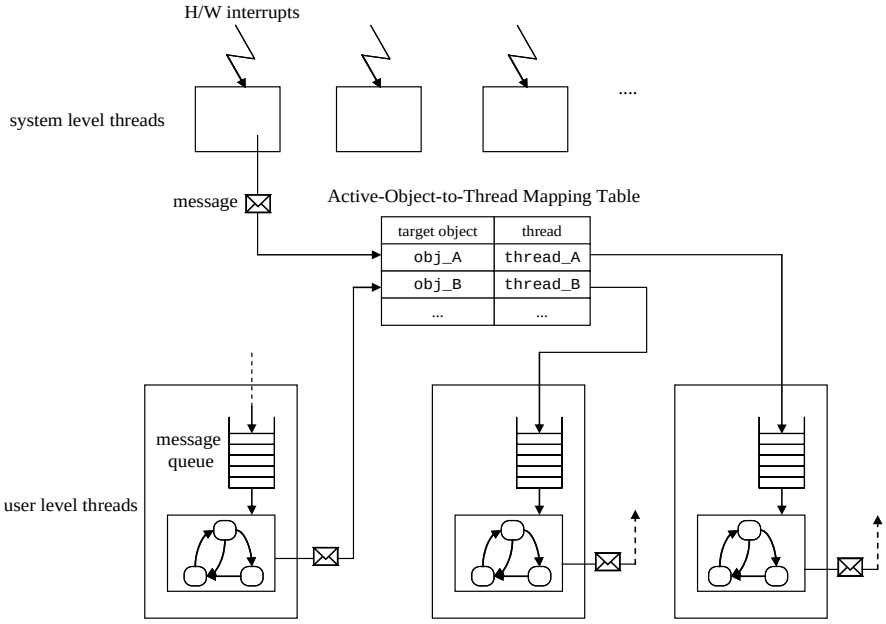

Figure 9. The architecture of the run-time system provided by RoseRT.

scenarios. Since this process is done perfunctorily, we omit a description of it here and append the actual algorithm in Appendix A. The latter provides an environment where developers can assign priorities to scenarios as needed and bind multiple scenario groups to be executed in a single task. Also, this tool automatically assigns suitable priorities to those scenarios where developers have failed to do so. Finally, the code modifier modifies the code generated by the code generator so that it can operate according to the run-time architecture of SISA. This tool operates via scenario and scenario group data provided by the scenario modeling tool.

The two newly added tools exchange data for scenarios and scenario groups through files in XML format. Figure 8 shows such an example XML file. After the scenario extractor generates this XML file, the scenario editor creates a graphic representation of this file and lets developers additionally insert data to this file. In Figure 8, each element `scenarioGroup` represents a scenario group and is uniquely identified by the attribute `id`. Developers can assign a priority to a scenario corresponding to a path from the root node to a specific activity node by assigning a priority to the final activity node. Then, the value of the attribute `priority` in the element `activityNode` that represents an activity node is recorded as the value determined by developers. In addition, developers can bind multiple scenario groups. For each binding, one element `MC` is created representing the bound scenario groups, and `id` values referring to the bound scenario groups are added as children of the element `MC`.

In SISA, a predecessor scenario must not have a lower priority than a successor scenario. That is, when represented as a scenario tree, a node should never have a lower priority than its child nodes. This is because if a predecessor scenario is run with a priority lower than its successor, then the successor is forced to run with a priority lower than its assigned priority. If the assigned priorities of some nodes conflict in such a way, the scenario editor raises the priority of the upper node so that it is equal to the highest priority among all of the lower nodes. For activity nodes that have not been assigned any priorities, the scenario editor assigns the same priorities as those of their parent or child nodes to satisfy this condition.

4.2. Run-time System Architecture

In this section, we discuss the SISA run-time system architecture. We begin by discussing the RoseRT run-time system architecture because it is the basis for our own. Following that, we explain how the RoseRT run-time architecture was modified to support SISA.

4.2.1. Run-Time System Architecture of RoseRT

Figure 9 shows the run-time system architecture of RoseRT. A system is composed of user level threads and system level threads. User level threads are threads that execute the state machines of active objects, and the system level threads are threads that are

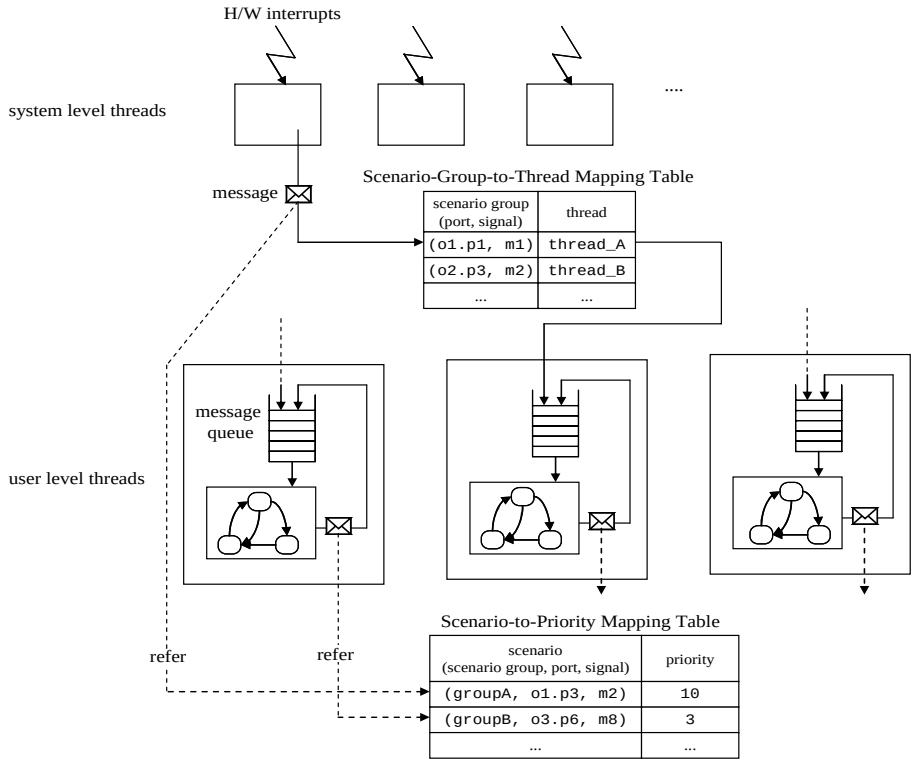

Figure 10. The architecture of the run-time system to support SISA.

awakened by hardware interrupts and deliver messages to the SPPs of active objects. Each user level thread has a message queue for buffering received messages.

When a system level thread is sending a message to a specific active object, it should find the user level thread that is executing the active object. For this, the active-object-to-thread mapping table is used. It is a table that maps an active object to the user level thread where it is executing. Once the user level thread is found, the message is stored in the per-thread message queue, and the thread is scheduled to run. Upon running, the thread extracts the message with the highest priority from its message queue. According to the kind of the message and the current state of the active object, an activity is selected and executed. Later on, the active-object-to-thread mapping table is referred to again when the thread wants to send a message to another active object.

This architecture can be optimized using several methods as follows. First, we can use separate queues for inter-thread message passing and intra-thread message passing. This reduces blocking times that can occur when message queues are accessed by multiple threads during inter-thread message passing. Additionally, we can use priority queues or bitmap queues to speed up the operations on the per-thread message queues. We do not explain such additional optimization schemes as they are out of the scope of the paper.

4.2.2. Modified Run-time System Architecture for SISA

Figure 10 shows the run-time system architecture devised to support SISA. Compared to the architecture in RoseRT, the main difference between two is that this architecture does not use the active-object-to-thread mapping table but instead uses the scenario-group-to-thread and scenario-to-priority mapping tables and IIP. Specifically, the following items are added or modified.

- When a message is created either by a system level thread or a user level thread, the priority of the message is determined by the scenario-to-priority mapping table. Each thread runs with the highest priority out of the pending messages and the message that is currently being processed.
- When system level thread sends a message to user level thread, the scenario-group-to-thread mapping table is used.
- When a user level thread sends a message, the message is always delivered to itself.
- While an activity is being executed, active objects and task 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

```
1    SAP-SEND(signal, data)
2        ▷this corresponds to SAP.
3        destinationPort ← connectedTo[this]
4        ▷(port, signal) corresponds to a scenario group.
5        scenarioGroup ← (destinationPort, signal)
6        userThread ← GETTHREAD(scenarioGroup)
7        ▷(scenario group, port, signal) corresponds to a scenario.
8        priority ← GETMESSAGEPRIORITY(scenarioGroup, destinationPort, signal)
9        ▷Creates a new message.
10       message ← (destinationPort, signal, data, priority, scenarioGroup)
11       ENQUEUE(queue[userThread], message)
12       if userThread is idle
13          SETPRIORITY(userThread, priority)
14       else
15          SETPRIORITY(userThread, max(priority, priority[userThread])
```

Figure 12. Pseudo code for sending a message from a system level thread to a user level thread.

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 the threads is obtained from the XML file provided by the scenario modeling tool. Before anything else, a thread is created for each scenario group that is not bound with other scenario groups. Then, a thread is created for each bundle of bound scenario groups.

Second, when the system is initialized, mutexes for all shared resources are created and initialized. Shared resources encompass all active objects and user level threads in the system. In accordance with IIP, the ceiling value of each mutex is initialized as the maximum of the priorities of all scenarios accessing the mutex. The information regarding which scenarios access which shared resource can be obtained perfunctorily by analyzing the XML file.

Third, when a system level thread generates a message, it should perform the following two sub-steps. The first sub-step is to find a user level thread to process the generated message and to send the message. After that, it performs the second sub-step that is to adjust the priority of the receiving thread to let the thread actually process the message. For the first sub-step, we use the scenario-group-to-thread mapping table. This table stores each scenario group and its executing thread. Since scenario groups are differentiated by the kinds of external messages arriving to specific SPP's, these are chosen as the keys of this table. Lines 5-6 in Figure 12 are for this sub-step, where the user level thread (*userThread*) to execute the scenario group corresponding to the message to be sent (*scenarioGroup*) is found using as keys the name of the SPP that will receive the message (*destinationPort*) and the message kind (*signal*).

The second sub-step involves making a thread run with the priority of the newly released scenario. Lines 8 and 12-15 in Figure 12 are for this sub-step, where the priority of the newly released scenario is determined by referring to the scenario-to-priority table. This table records each scenario and the priority with which the scenario will be executed. Since scenarios are differentiated by which ports will receive which kinds of messages in which scenario groups, these are chosen as the keys of this table. After the priority (*priority*) is determined by referring to the mapping table, the priority of the user level thread (*userThread*) is set depending on whether the

```
1    MAINLOOP
2        forever
3           ▷A message with highest priority is dequeued.
4           ▷When there is no message in the queue, thread is blocked.
5           ▷this corresponds to a thread currently running.
6           message ← DEQUEUE(queue[this])
7           destinationObject ← owner[destinationPort[message]]
8           SETPRIORITY(this, max(ceiling[destinationObject], ceiling[this]))
9           ▷ State is changed, then an activity is selected and executed.
10          RUNFSM(this, message)
11          SETPRIORITY(this, (max(for all priorities of messages in queue[this])
```

Figure 11. Pseudo code for processing messages in user level threads.

```
1    PORT-SEND (signal, data)
2        ▷this corresponds to a port object that is used to send a message.
3        destinationPort ← connectedTo[this]
4        scenarioGroup ← scenarioGroup[currentMessage[owner[this]]]
5        priority ← GETMESSAGEPRIORITY(scenarioGroup, destinationPort, signal)
6        message ← (destinationPort, signal, data, priority, scenarioGroup)
7        ▷currentThread refers to a thread currently running.
8        ENQUEUE(queue[currentThread], message)
```

Figure 13. Pseudo code for sending messages in user level threads.

thread is already executing any other scenario or not. First, if the thread is in the idle state where it is not currently executing any scenarios, then the priority of the thread is set to the priority determined by the mapping table (*priority*). Otherwise, the priority is set to the larger value between the current priority with which the thread is executing (*priority*[*userThread*]) and the priority determined by the mapping table (*priority*).

Forth, the user level thread that receives the message fetches the highest priority message from its message queue and then executes an activity according to its state machine. To support SISA, this step entails 1) protecting the active object and the thread context using IIP before an activity is executed, and 2) adjusting the priority of the thread after the execution of an activity finishes. The former is done by line 8 of Figure 11, where the priority of the thread is increased to the maximum of the mutex ceilings for the active object receiving the message (*destinationObject*) and the thread (*this*). As such, the protection of shared resources is implemented by increasing the priorities of threads in IIP. If the thread priority is raised, those scenarios that may access this active object and these threads cannot start and thus synchronization is achieved automatically.

The latter is done by line 11 of Figure 11, where the thread priority is set to the priority of the message that has the highest priority among the messages in the current message queue (*queue*[*this*]). This accomplishes the following two goals simultaneously: it not only releases the protection for the active object and the thread context, but also adjusts the thread priority to the priority of the scenario that will be executed the next time.

Finally, when a user level thread executes an activity of an active object, it may send messages to other objects. Figure 13 shows the pseudo code for sending a message in user level threads. Line 5 is for determining the priority of a message referring to the scenario-to-priority mapping table just like when a system level thread sends a message. Line 8 is for inserting the generated message to the message queue of a thread. Note that the thread that owns the message queue is the currently executing user level thread (*currentThread*). That is, the thread sending a message and the thread receiving the message are always the same.

5. Experimental Performance Evaluation

In this section, we evaluate the real-time performance of code generated by SISA. We compare the performance of our code with RoseRT-generated code that uses task sets derived by the best known existing method. Specifically, we used the task set derivation method proposed in [10], [11], [14]. As a target system, we used an industrial private branch exchange (PBX) system. This system is connected with a large group of cell phones and is designed to service their call requests. The PBX system used in the experiments is a complicated embedded system whose structure and behavior dynamically change to an extreme degree. This system repetitively creates and destroys active objects corresponding to serviced phones, call requests, call connections, etc. at run time. We first explain the environmental setup and the performance metrics, then present the performance results.

5.1. Experimental Setup

The PBX system is composed of active objects abstracting the cell phones, system administrators, etc. as well as active objects for handling each call request, call connection, connection monitoring, etc. The actors that use the system include cell phones, a timer, system administrators, etc. Scenarios are released when the system receives external messages from these actors. The external messages initiating scenarios in the PBX system are 1) button (power, numbers, send, and end) messages, 2) timeout messages for periodically sending ping messages to each phone to monitor its connection status, 3) acknowledgement messages sent by each phone for these ping messages, and 4) messages sent by system administrators to monitor the status of the PBX system or to modify the system configuration. We have assigned priorities to such external input messages (scenarios) according to the following rules.

- All button messages sent from one cell phone have the same priority
- Each cell phone has a different priority, and each button message has the priority of its sending phone.
- Messages sent from system administrators have priorities lower than those of the button messages for all phones.
- The timeout messages for generating ping messages and acknowledgement messages have the lowest priority.

The environment used for the experiments is as follows. The target hardware is a Sun Blade 100 and the operating system is Sun Solaris 9 (SunOS 5.9). The library that supports the abstraction layer for the hardware and operating system is the RoseRT run-time system library 2003.06.00 compiled with gcc 3.0.1. When we perform the experiments, we adjusted the number of cell phones connected to the PBX system between 5 and 100.

5.2. Performance Metrics

Our performance metrics are as follows.

- The maximum scenario blocking time
- The maximum scenario response time
- The standard deviation of the averaged scenario blocking times in systems with various numbers of scenarios
- The standard deviation of the response times of mission critical scenarios in systems with various numbers of scenarios
- The number of derived tasks

The first two metrics determine the responsiveness of the code. Scenario blocking time is the amount of time that a scenario waits while lower or same priority scenarios are executing. Scenario response is the amount of time from when an external message triggering the scenario is inserted into the thread message queue until the final activity of the scenario finishes its execution. This time period is composed of 1) the inter-thread message passing time, 2) the inter-thread context switching time, 3) the summation of the execution times of the activities constituting the scenario, 4) the blocking time by lower or same priority scenarios, and 5) the interference time spent while the scenario is preempted by higher priority scenarios.

The last three metrics determine whether the code can support large-scale systems or not. The blocking times of all scenarios should not drastically increase as the number of scenarios increases so that the code can work properly for large-scale systems. On the other hand, if the system becomes larger, the response times of the lower priority scenarios inevitably become longer as the number of the higher priority scenarios increases. However, the response time of the mission critical scenarios should be almost constant regardless of the scale of the system so that the system can respond to mission critical events on time. To quantify such system features, we calculate the standard deviations for the averaged scenario blocking times as the number of scenarios increases. We also do so for the response times of mission-critical scenarios. The last metric, the number of tasks, also should be as small as possible even for large-scale systems so as to not only reduce context switching overhead but also to save memory.

5.3. Performance Results

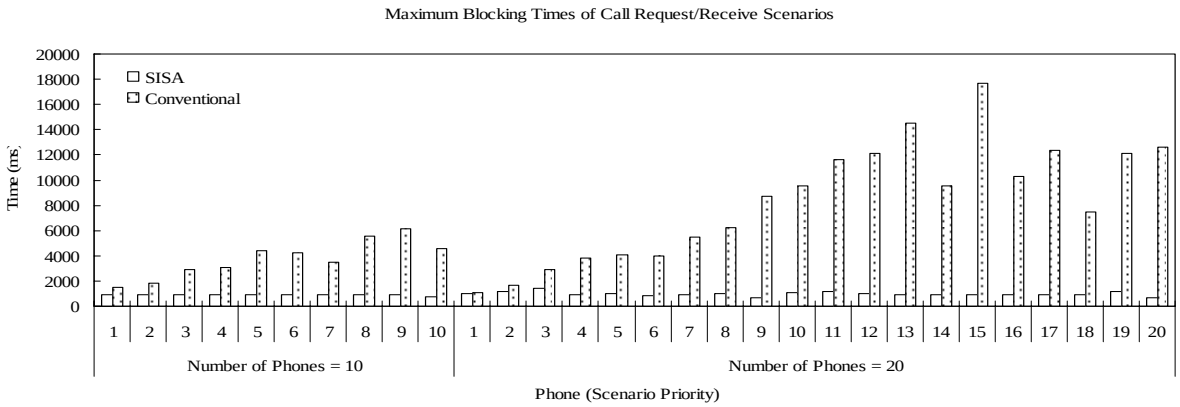

Figure 14. Maximum blocking times for call request/receive scenarios.

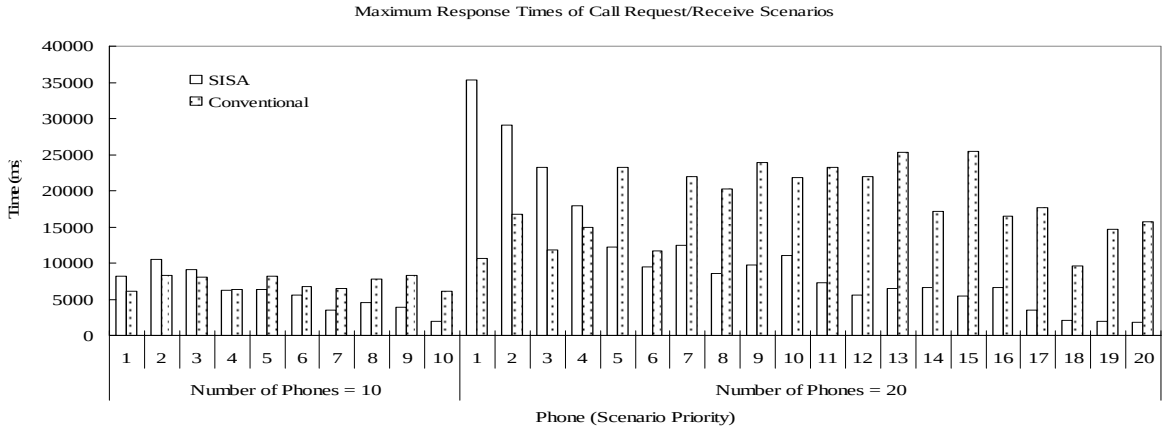

Figure 15. Maximum response times for call request/receive scenarios.

We present the results for call request/receive scenarios among the various scenarios in the system. These are scenarios that are initiated when the send buttons of cell phones are pressed. We do not present all results since the other scenarios also showed similar results. Throughout this section, scenarios are numbered in ascending order as their priorities get higher.

1. Blocking Time: Figure 14 shows the maximum blocking time of each scenario when the number of phones is 10 and when t is 20. The results show that the maximum blocking times of all scenarios are greatly decreased in SISA compared to the conventional case. Specifically, for the system with 20 phones, the maximum blocking times of all scenarios were reduced 79.2% on average. Moreover, the maximum blocking times of scenarios in SISA are almost constant regardless of scenario priorities while those in the conventional case increase as the scenario priorities become higher. This is because scenarios in the conventional case experience multiple blockings and the higher a scenario's priority is, the more blockings there are, due to a larger number of scenarios with relatively low priority. Therefore the blocking time for higher priority scenarios is longer. As a result, we were able to reduce the blocking time of the mission critical scenario (the 20th phone) by 94.5%.

2. Response Time: Figure 15 shows the maximum response time of each scenario when the number of phones is 10 and when it is 20. We can see that the response times of all scenarios in SISA are generally smaller than those in the conventional implementation. Specifically, for the system with 20 phones, the response times of all scenarios were reduced by about 30.3% on average. Also, higher priority scenarios have far shorter response times in SISA compared to those in the conventional approach, while for lower priority scenarios the opposite is true. For the system with 20 phones, the response time of the mission critical scenario (the 20th phone) was decreased 88.6% in SISA, while the response time of the lowest priority scenario (the first phone) was increased more than two times (230%). This is because the lower priority scenarios in SISA experience more interference time since the higher priority scenarios are less blocked and thus preempt the lower priority scenarios more frequently. That is, the response times of the lower priority scenarios in SISA cannot avoid becoming longer as the response times of the higher priority scenarios become shorter.

3. The Number of Derived Tasks and Blocking/Response Time Variation According to the Increase in the Number of Scenarios: Figure 14 shows that in SISA the blocking times of all scenarios are almost constant regardless of the number of phones, while in the conventional case the blocking times in the system with 20 phones were obviously larger than those in the system with 10 phones. Also in Figure 15, if we compare the response times of the mission critical scenarios in the two systems (the 10th phone and the 20th phone),

	SISA	Conventional approach
Blocking times of all scenarios	99.78 ms	3324.93 ms
Response times of mission-critical scenarios	199.18 ms	6084.52 ms

Table 1. Standard deviations of blocking/response times according to the increase in the number of scenarios.

they are almost the same when SISA is used, while in the conventional case the value is greatly increased in the system with 20 phones.

To quantify these phenomena, we calculated the standard deviations for 1) the average blocking times of all scenarios and 2) the response times of mission critical scenarios in several systems with differing numbers of scenarios. The results in Table 1 show that the deviation in the blocking times of scenarios according to the varying number of scenarios was decreased 97% in SISA, and the deviation in the response times of mission-critical scenarios was also decreased by 98%.

On the other hand, the number of derived tasks was as follows with n being the number of phones.

- $n + 0.5n + 10$ tasks in the conventional approach. The first n tasks are for active objects that abstract phones. The second $0.5n$ tasks are for active objects that handle each call connection. The last 10 tasks are for all remaining active objects such as the object that abstracts the system administrators, the object for connection monitoring, and several controller and database-manager objects.
- $n + 2$ tasks in SISA. The first n tasks are for scenarios handling phone requests. The second 2 tasks are for the scenario that handles requests from the system administrators and the scenario that monitors the connection status.

As such, the number of tasks derived using the conventional approach was much larger than when SISA was used. These results clearly show that SISA supports large-scale systems better than the conventional approach. Moreover, these results also show that the performance difference between SISA and the conventional approach increases as the scale of the systems grows.

6. Related Work

There have been many research efforts to improve the performance of code automatically generated via the active-object-based task set derivation method. First, there has been some work to present guidelines for grouping active objects to improve code performance [10], [11], [14], [35]. However, as we have explained in Section 1.1, these guidelines are difficult for developers to apply in practice. Even if they are applied, they are limited in that they are not a real means of solving the real-time performance problem. Additionally, there have been many efforts to help developers predict code performance by helping them analyze the schedulability of a system designed with active objects. [38] presented a method that analyzes the response time for each instance of message processing. Schedulability analysis algorithms were presented by [13] using utilization analysis and by [35] and [37] using response time analysis. Recently, [6], [15], [18] and [34] have presented how to exploit existing real-time schedulability analysis algorithms when models are designed using real-time UML profile [29]. On the other hand, [36] and [37] have presented an idea that uses preemption threshold scheduling [44] to limit blocking to a single occurrence until the system responds. However, it is based on the assumption that the priorities of all messages are fixed. That is, it is limited in that scenarios themselves cannot have priorities since messages cannot change priority according to the external messages which trigger them. Moreover, this work has left mapping messages to tasks as an open problem, and has not implemented the idea nor performed any experimental evaluation.

SISA was inspired greatly by scenario-based design and requirement engineering. As [8] has presented a survey of scenario-based design and [45] has presented current practices for how scenarios are used in developing systems, there have been many research activities introducing scenarios to complement object-oriented design. Most work uses scenarios to model requirements and goals as in [8], [22], [31], and [32] or focuses on synthesizing behavior models from scenarios as in [24], [25], [43], and [46]. SISA focuses on automatically generating multitasking implementations where each scenario is mapped to an implementation-level task. As such, our specific purpose for using scenarios is different, but the motivations are the same in that scenarios are treated as the modeling entity that is recognizable to end users and is directly related to their requirements such as timing constraints.

SISA was also influenced by many research activities that provide methods to manage multiple scenarios. Real-time UML profiles [29] and high-level MSC (hMSCs) [1], [33] provide a way to compose a scenario by relating multiple scenarios. [9] and [42] also presented a way to implicitly associate scenarios using preconditions or triggering conditions, where developers designate when each scenario can be triggered instead of directly associating multiple scenarios. Like this work, SISA derives scenarios and scenario groups by identifying external messages and provides a way to aggregate multiple scenario groups to map the same task.

At the same time, SISA was influenced by various model transformation methods. Much research has been done on model transformation methods to predict system performance [3], [5] while [12], [24], [28], and [40] use intermediate models to produce better customized output. Similarly, SISA uses scenario trees as intermediate models for users to additionally design the timing characteristics of the system. On the other hand, [12], [17], [24], and [27] proposed a method to generate a model transformer by specifying the rules for meta-model transformation. These methods can be integrated to SISA when scenario trees are derived from a network of active objects.

7. Conclusion

We have presented SISA, an architecture consisting of a method for deriving task sets, the supporting development tools, and a run-time system architecture that together allow developers to produce optimal response times in object-oriented embedded systems. This paper makes four main contributions. First, we proposed the scenario-based task set derivation method to derive a task set that has a minimal response time for each scenario and consists of the smallest possible number of tasks. This method eliminates blocking due to inter-task message passing and ensures that blocking time is less than the longest execution time of all the activities of lower priority scenarios. Second, we presented development tools so that developers can represent the timing characteristics of the system without modifying the object-oriented model of the system, which was impossible with existing development tools. Third, we proposed a run-time system architecture to support the proposed method. This architecture clearly shows how our method can be supported by modifying or adding some features to the conventional architecture. Finally, we applied SISA to an existing industrial PBX system and presented the results of our performance evaluations. These results clearly show that SISA drastically improves performance by reducing the blocking time of the mission critical scenario by 94.5% compared to the best known conventional approach. The results of our experiments also show that SISA supports large-scale systems better than conventional approaches.

We are actively expanding this research in a number of directions. First, we are considering how to effectively maintain consistency between the two modeling views, that is, between the network of active objects and the set of scenarios. Specifically, when the original system designed with active objects is changed, we should provide a way to minimize the additional work required to reflect the changes to the scenario trees. Second, we are currently performing research on how to automatically generate code that guarantees real-time schedulability by applying advanced real-time scheduling mechanisms which extend fixed priority scheduling. If this becomes possible, developers can simply model deadlines instead of scenario priorities and meet these deadlines with automatically generated code. Third, we need to extend SISA to support distributed systems. To begin with, we are considering how to support replaceable active objects from different binaries without recompiling. Also, we are conducting research on how SISA can be applied to distributed systems composed of multiple processes. Finally, we are investigating the specific performance benefits achieved by SISA in various real-time embedded applications including automotives and mobile robots.

References

[1] R. Alur, G.J. Holzmann, and D. Peled, An analyser for message sequence charts, Proceedings Second Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96), 1996, pp. 35-48.

[2] ARTiSAN Software Tools Incorporation. Real-Time Studio, http://www.artisansw.com.

[3] A. D'Ambrogio, A model transformation framework for the automated building of performance models from UML models, Proceedings of the International Workshop on Software and Performance, 2005, pp. 75-86.

[4] M. Awad, J. Kuusela, and J. Ziegler, Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion, Prentice Hall, 1996.

[5] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, Model-based performance prediction in software development: a survey, IEEE Transactions on Software Engineering, vol. 30, no. 5, 2004, pp. 295-310.

[6] A. J. Bennett and A. J. Field, Performance engineering with the UML profile for schedulability, performance and time: A case study, Proceedings of IEEE Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, October 2004, pp. 67-75.

[7] A. Burns and A. Wellings, Real-Time Systems: Specification, Verification, and Analysis, chapter Advanced Fixed Priority Scheduling, Prentice Hall, 1996, pp. 32-65.

[8] J. M. Carroll, R. L. Mack, S. P. Robertson, and M. B. Rosson, Binding objects to scenarios of use, International Journal of Human–Computers. Stud., vol. 41, no. 1/2, 1994, pp. 243–276.

[9] J. M. Carroll, Ed., Scenario-Based Design: Envisioning Work and Technology in System Development, John Wiley and Sons, 1995.

[10] B. P. Douglass, Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns, Addison-Wesley, 1999.

[11] B. P. Douglass, Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley, 1999.

[12] D. Galran, L. Cai, and R. L. Nord, A transformational approach to generating application specific environments, Proceedings ACM SIGSOFT Symposium of Software Development Environment, 1992, pp. 68-77.

[13] D. Gaudrean and P. Freedman, temporal analysis and object-oriented real-time software development: a case study with ROOM/Objectime, Proceedings of IEEE Real-Time Systems Symposium, 1996, pp. 110-119.

[14] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley Longman, 2000.

[15] G. P. Gu and D. C. Petriu, Early evaluation of software performance based on the UML performance profile, Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, 2003, pp. 66-79.

[16] G. Gullekson and B. Selic, Design patterns for real-time software, Proceedings of Embedded Systems Conference West, 1996.

[17] W. Ho, J. Jézéquel, A. Guennec, and F. Pennaneac'h, UMLAUT: An extendible UML transformation framework, Proceedings of Automated Software Engineering (ASE'99), 1999, pp. 275-278.
[18] D. Jin and D. C. Levy, An approach to schedulability analysis of UML-based real-time systems design, Proceedings of the International Workshop on Software and Performance, 2002, pp. 243-250.
[19] IAR Systems Incorporation, visualSTATE, http://www.iar.com
[20] IBM Rational Software Corporation, Rational Rose RealTime User Guide: Revision 2001.03.00, 2000.
[21] I-Logix Incorporation. Rhapsody tools. http://www.ilogix.com
[22] Institute for Electrical and Electronic Engineers and The Open Group, Base Specifications Issue 6, IEEE Std. 1003.1 (POSIX), 2004 Edition, System Interfaces volume, 2004.
[23] H. Kaindl, A design process based on a model combining scenarios with goals and functions, IEEE Transactions on Systems, Man, and Cybernetics—Part A: Systems and Humans, vol. 30, no. 5, 2000, pp. 537-551.
[24] G. Karsai, J. Sztipanovits, and H. Franke, Towards specification of program synthesis in model-integrated computing, Proceedings IEEE ECBS Conference, 1998, pp. 226-233.
[25] I. Kru¨ ger, R. Grosu, P. Scholz, and M. Broy, From MSCs to statecharts, Distributed and Parallel Embedded Systems, F.J. Rammig, ed., Kluwer Academic Publishers, 1999, pp. 61-71.
[26] A. Lamsweerde and L. Willemet, Inferring declarative requirements specifications from operational scenarios, IEEE Transactions on Software Engineering, vol. 24 , no. 4, 1998, pp. 1089 -1114.
[27] D. Milicev, Automatic model transformations using extended UML object diagrams in modeling environments, IEEE Transaction on Software Engineering, vol. 28, no. 4, 2002, pp. 413-431.
[28] J. Mukerji and J. Miller, Model Driven Architecture (MDA) Guide Version 1.0.1, OMG Document Number: omg/2003-06-01, 2003.
[29] Object Management Group, UML Profile for Schedulability, Performance, and Time, OMG Document ptc/2003-09-01, http://www.omg.org/cgi-bin/doc?ptc/2002-09-03, 2003.
[30] Object Management Group. Unified Modeling Language (UML), version 2.0, OMG Documentation, http://www.omg.org/technology/documents/formal/uml.htm, 2005.
[31] C. Potts, Using schematic scenarios to understand user needs, Proceedings of Symposium on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS ’95). MI: ACM, August 1995, pp.247–256.
[32] C. Rolland, C. Souveyet, and C. Ben Achour, Guiding goal modeling using scenarios, IEEE Transactions on Software Engineering, vol. 24, 1998, pp. 1055–1071.
[33] E. Rudolph, P. Graubmann, and J. Grabowski, Tutorial on message sequence charts, Proceedings IFIP TC6 WG6.1 Int’l Conf. Formal Description Techniques (FORTE), 1996, pp. 1629-1641.
[34] H. Saiedian and S. Raghuraman, Using UML-based rate monotonic analysis to predict schedulability, IEEE Computer, vol. 37, no. 10, October 2004, pp. 56-63.
[35] M. Saksena, P. Freeman, and P. Rodziewicz, Guidelines for automated implementation of executable object oriented models for real-time embedded control systems, Proceedings of IEEE Real-Time Systems Symposium, 1997, pp. 240-251.
[36] M. Saksena, P. Karvelas, and Y. Wang, Automatic synthesis of multi-tasking implementations from real-time object-oriented models, Proceedings of IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2000, pp. 360-367.
[37] M. Saksena and P. Karvelas, Designing for schedulability: integrating schedulability analysis with object-oriented design, Proceedings of Euromicro Conference on Real-Time Systems, 2000, pp. 101-108.
[38] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz, Schedulability analysis for automated implementations of real-time object-oriented models, Proceedings of IEEE Real-Time Systems Symposium, 1998, pp. 92-102.
[39] B. Selic, G. Gullekson, and P. T. Ward, Real-Time Object-Oriented Modeling, John Wesley and Sons, 1994.
[40] C. Simonyi, Intentional Programming-Innovation in the Legacy Age, International Federation for Information Processing Work Group 2.1, http://research.microsoft.com/ip, June, 1996.
[41] Telelogic Corporation. TAU, http://www.telelogic.com
[42] P.P. Texel and C.B. Williams, Use Cases Combined with Booch, OMT, and UML, Prentice-Hall, 1997.
[43] S. Uchitel, J. Kramer, and J. Magee, Synthesis of behavioral models from scenarios, IEEE Transactions on Software Engineering, vol. 29, no. 2, 2003, pp. 99-115.
[44] Y. Wang and M. Saksena, Scheduling fixed priority tasks with preemption threshold, Proceedings of IEEE Real-Time Computing Systems and Applications Symposium, 1999, pp. 328-335.
[45] K.Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, Scenarios in system development: current practice, IEEE Software, 1998, pp. 33-45.
[46] J. Whittle and J. Schumann, Generating statechart designs from scenarios, Proceedings of International Conference on Software Engineering (ICSE ’00), 2000, pp. 314-323.
[47] S. S. Yau and Z. Xiaoyong, Schedulability in model-based software development for distributed real-time systems, Proceedings of the International Object-Oriented Real-Time Dependable Systems, 2002, pp. 45-52.
[48] J. Zalewski, Real-time software architectures and design patterns: fundamental concepts and their consequences - keynote address, Proceedings of 24th IFAC/IFIP Workshop on Real-Time Programming, 1999.

Appendix A. Scenario Tree Generation Algorithm

Figure 16 shows the algorithm for generating scenario trees. In this algorithm, a scenario tree corresponds to a scenario group and generating a scenario tree corresponds to partitioning scenarios into a group. Our method is based on reachability analysis where scenarios are partitioned into the same group when their activities can be reached from the same external events. This algorithm first 1) generates a scenario tree from each external message arriving at an SPP (TOTAL-TREES-BUILD()) and then 2) traces the paths of the message flows considering all possible generation of messages (TREE-BUILD-FROM-MESSAGE/ACTIVITY/JUNCTION-NODE()). Note that tracing the paths of possible message flows is done by deriving a regular expression for messages sent by each activity in TREE-BUILD-FROM-ACTIVITY-NODE() and TREE-BUILD-FROM-JUNCTION-NODE(). The complexity of the algorithm is $O(m \cdot n)$ where m is the number of incoming messages from SPP ports and n is the maximum number of activity nodes in trees.

```
TOTAL-TREES-BUILD(M) ▷ M is the input model.
▷ Trees is the set of all scenario trees
1  Trees ← {}
2  for each p ∈ SPPs[M]
3     do for each m ∈ IncomingMessages[p]
4        do create tree T
7           create message node n with message m
8           root[T] ← n
9           TREE-BUILD-FROM-MESSAGE-NODE(n)
10          Trees ← Trees ∪ {T}

TREE-BUILD-FROM-MESSAGE-NODE(n)
1  A ← MatchesActivities[n]
2  if |A| = 1
3     then create activity node n' with activity a ∈ A
4        child[n] ← n'
5        TREE-BUILD-FROM-ACTIVITY-NODE(n')
6  else create disjunction node n'
7     child[n] ← n'
8     for each a ∈ A
9        do create activity node n" with activity a
10          child[n'] ← n"
11          TREE-BUILD-FROM-ACTIVITY-NODE(n")

TREE-BUILD-FROM-ACTIVITY-NODE(n)
1  P ← REGULAR-EXPRESSION-FOR-MESSAGES-SENT-BY(activity[n])
2  P' ← COMPONENTS-WITHOUT-OUTERMOST-JUNCTIONS(P)
3  if |P'| = 1  ▷ only one message is sent out unconditionally.
4     then create message node n' with message m ∈ P'
5        child[n] ← n'
6        TREE-BUILD-FROM-MESSAGE-NODE(n')
7  elseif |P'| ≠ 0 ▷ one or more messages are sent out conditionally.
8     then create junction node n' of type OUTERMOST-JUNCTION-TYPE(P)
9        child[n] ← n'
10       for each p ∈ P'
11          do TREE-BUILD-FROM-JUNCTION-NODE(n', p)

▷ P is a regular expression for message sequence.
TREE-BUILD-FROM-JUNCTION-NODE(n, P)
1  P' ← COMPONENTS-WITHOUT-OUTERMOST-JUNCTIONS(P)
2  if |P'| = 0  ▷ no message is sent out.
3     then create flow final node n'
4        child[n] ← n'
5  elseif |P'| = 1 ▷ only one message is sent out unconditionally.
6     then create message node n' with message m ∈ P'
7        child[n] ← n'
8        TREE-BUILD-FROM-MESSAGE-NODE(n')
9  else  ▷ One or more messages are sent out conditionally.
10    then create junction node n' of type OUTERMOST-JUNCTION-TYPE(P)
11    child[n] ← n'
12       for each p ∈ P'
13          do TREE-BUILD-FROM-JUNCTION-NODE(n', p)
```

Figure 16. Algorithm for generating scenario trees.