Modeling Fixed-Priority Preemptive Multi-Task Systems in SpecC

Tomiyama, Hiroyuki
Institute of Systems & Information Technologies/KYUSHU

Cao, Yun
Graduate School of Information Science and Electrical Engineering

Murakami, Kazuaki
Graduate School of Information Science and Electrical Engineering

出版情報：Proc.of IEEE The 10th Workshop on System And System Integration of Mixed Technologies (SASIMI'01), pp.93-100, 2001-10. Workshop on System And System Integration of Mixed Technologies
バージョン：
権利関係：

Modeling Fixed-Priority Preemptive Multi-Task Systems in SpecC

Hiroyuki Tomiyama [†]
tomiyama@isit.or.jp

Yun Cao [†‡]
cao@c.csce.kyushu-u.ac.jp

Kazuaki Murakami [†‡]
murakami@i.kyushu-u.ac.jp

[†] Institute of Systems & Information Technologies/KYUSHU
2–1–22–707 Momochihama, Sawara-ku, Fukuoka 814–0001, Japan

[‡] Graduate School of Information Science and Electrical Engineering
Kyushu University
6–1 Kasuga-Koen, Kasuga 816–8580, Japan

Abstract

Many real-world embedded systems employ a preemptive scheduling policy in order to satisfy their real-time requirements. However, most System-Level Design Languages (SLDLs) which were proposed up to now, such as SpecC, do not explicitly support modeling of preemptions. This paper proposes techniques for modeling fixed-priority preemptive multi-task systems in the SpecC SLDL. The modeling techniques with SpecC enable a system designer to specify and simulate preemptive multi-task systems including both software and hardware at a high level of abstraction, without assuming any specific real-time operating system.

Keywords: *SpecC, system-level modeling, preemptive systems.*

1 Introduction

Due to the continuously increasing number of transistors on a chip as well as the growing complexity of embedded software, it is widely agreed that in the near future we will need System-Level Design Languages (SLDLs) for efficient design of such complex hardware/software systems at high levels of abstraction. SpecC [4, 5] is one of the most promising SLDLs. It was originally developed at University of California, Irvine (UCI) in 1997 [5]. In 1999, SpecC Technology Open Consortium (STOC) [2] was founded, and since then STOC has been working for standardization of the SpecC language.

Syntactically, SpecC is an extended ANSI-C language. Besides all the features supported by ANSI-C, SpecC further supports several features commonly exhibited in many embedded systems such as structural hierarchy, behavioral hierarchy, concurrency, pipelined execution, state transition, exception handling, and so on. Due to these extended features of SpecC, a designer can seamlessly describe both hardware and software at different levels of abstraction, from the specification to implementation level (RT-level for hardware, and C-level for software), all in the same language. Thus, SpecC drives hardware/software co-design, co-simulation and co-verification.

Although, as an SLDL, SpecC is superior to most of existing HDLs (e.g., VHDL and Verilog) or software programming languages (e.g., C, C++, and Java), some important features are still missing. One of such features is preemptive task scheduling.

Many real-world embedded systems are reactive real-time systems where tasks are invoked by events from their external environment (e.g., sensors) and must be completed within their predefined deadline time. In order to better utilize the processor resource to meet the deadlines, preemptions are often allowed. In such preemptive systems, a task which is currently under execution can be preempted by a higher-priority task when it gets ready to run.

In spite of the effectiveness and the popularity of preemptive scheduling, SpecC does not explicitly support modeling of preemptions. In the final implementation, preemptive scheduling is usually realized with services by Real-Time Operating Systems (RTOSs). At an early design phase, however, the system specification should not depend on a specific RTOS. Therefore, we definitely need techniques to capture preemptive systems in SLDLs like SpecC without assuming any specific RTOS.

In this paper we propose techniques to model preemptive multi-task systems in the SpecC language. Using the modeling techniques, a designer can specify

preemptive multi-task systems at a high level of abstraction without considering implementation details such as RTOS-dependent system calls. Then, he/she can verify whether real-time constraints are satisfied through simulation. If the constraints are violated, he/she can explore the design space by changing tasks' priorities or hardware/software partitioning.

This paper is organized as follows. As preliminaries, Section 2 briefly describes two important features of the SpecC language, *concurrency* and *exception handling*, which are used to model preemptions in the rest of this paper. Then, Section 3 proposes two techniques to model fixed-priority preemptive multi-task systems in SpecC. Our preliminary experiments are presented in Section 4. Section 5 discusses some problems in the current version of the SpecC language which we have found through the experiments. Finally, Section 6 concludes this paper with a summary.

2 Concurrency and Exception Handling in SpecC

SpecC explicitly supports modeling of concurrency and exception handling. There are two types of concurrency in SpecC; *parallel execution* and *pipelined execution*. In this section, we only explain parallel execution in SpecC since pipelining will not be used in this paper. Figure 1 shows a simple example of parallel execution in SpecC. In this example, *Task* is defined as a **behavior** class, and three tasks are instantiated as sub-behaviors of the *Main* behavior. Then, they are executed in parallel using the **par** construct. Note that the execution order of concurrent behaviors is not defined in the SpecC language but is implementation-dependent.

In SpecC, two types of exception handling, *abortion* and *interrupt*, are supported. Figure 2 shows an example code of exception handling using the **try-interrupt-trap** construct. In the example, if event *e1* happens during execution of behavior *b0*, *b0* is immediately interrupted by *b1*. After *b1* is completed, execution of *b0* is resumed from the point of the interrupt. If event *e2* happens during *b0*, execution of *b0* is immediately terminated and is taken over by *b2*. After completion of *b2*, the control leaves the **try-interrupt-trap** statement and goes to the following line. Note that if both events *e1* and *b2* happen at the same time, *b1* is executed since it is listed first in the program text. Event *e2* is ignored. It should be also noted that any event is ignored during exception handling, unless another **try-interrupt-trap** statement is defined in the exception handling behavior.

For more details on the SpecC language, refer to the SpecC Language Reference Manual [3].

```
1:  behavior Task() { // sub-behavior
2:     void main() {
3:        for (;;) { // infinite loop
4:           waitfor(1);
5:        }
6:     }
7:  };
8:
9:  behavior Main() { // Main behavior
10:    Task task1, task2, task3;
11:    void main() {
12:       par {
13:          task1.main();
14:          task2.main();
15:          task3.main();
16:       }
17:    }
18: };
```

Figure 1. Parallel execution in SpecC.

```
1: event e1, e2;
2: try           { b0.main(); }
3: interrupt (e1) { b1.main(); }
4: trap (e2)      { b2.main(); }
```

Figure 2. Exception handling in SpecC.

3 Modeling Fixed-Priority Preemptions

In this section, we propose two techniques to model preemptive multi-task systems. We assume that in the systems multiple tasks run on a single processor based on a preemptive scheduling policy. The tasks may be either periodic or aperiodic. Priorities are statically given to the tasks, and are fixed during execution. Among all the tasks which are ready to run, only one task with the highest priority is executed on the processor at a time.

This section uses as a demonstrative example a preemptive system consisting of three tasks, *task1*, *task2* and *task3*, where priorities are given to them in this order (*task1* has the highest priority).

3.1 Preemption Modeling with Hierarchically Composed Exception Handling

In SpecC, exception handling with the **try-interrupt** construct is only the mechanism which realizes suspension and resumption of tasks' execution. Therefore, the simplest way to model preemptions is to define the higher-priority tasks as exception handlers of the lower-priority tasks. For our example system, *task1*

is defined as an exception handler of *task2*, while *task2* is in turn an exception handler of *task3*. In this way, this preemptive system is hierarchically composed of **try-interrupt** statements.

Figure 3 shows a fragment of a SpecC code which realizes preemptions. In the top-level behavior (the *Main* behavior, lines 31–43), three tasks are instantiated and defined as exception handlers. The behavior named *dummy* does not perform any meaningful computation. When an event happens, the corresponding task gets executed. Since *task1* is listed first in the **try-interrupt-interrupt-interrupt** statement (lines 38–41), it has the highest priority. Note that *task3* can be preempted by *task1* and *task2*, and *task2* can be preempted only by *task1*. Definition of *task1* is omitted in Figure 3. However, it should be mentioned that there must not be any exception handling statement in *task1* so that *task1* cannot be preempted by any other tasks.

Although this modeling technique might seem to well realize the fixed-priority preemptive scheduling, it suffers from two serious problems. One problem is that it does not correctly reflect the behavioral hierarchy of the system. With this modeling, higher-priority tasks are modeled as sub-behaviors of lower-priority tasks. This makes it difficult to analyze and optimize the designs in the later design phases. Furthermore, for each task (except the lowest-priority task), multiple instances are defined although only one instance is necessary in actual. This may lead to inefficient implementations. For example, in Figure 3, there exist three instances of *task1* and two instances of *task2*. If we compile this SpecC code in a straight-forward manner, there will be three copies of *task1* and two copies of *task2* in the final object code, or we need a run-time support to create and delete tasks on-the-fly. In either way, this modeling suffers from overhead of code size and/or performance. In addition, improper specification of behavioral hierarchy may result in unnecessary design errors in the later design phases. Obviously, we need better techniques to capture the behavioral hierarchy correctly and naturally.

The other problem of this modeling is more serious. In the modeling above, any event which arrives during execution of a higher-priority task is ignored. For example in Figure 3, if event *e2* is notified during execution of *task1*, the event is ignored, hence, an execution of *task2* is lost. Also, an event is lost when more than one event happen at the same time. In many real-world systems, however, losing events may result in a fatal error at run-time. Any event must be kept until the current task is completed, and then, the corresponding task must get executed.

```
1:  event e1; // activates task1
2:  event e2; // activates task2
3:  event e3; // activates task3
4:
5:  behavior Task1(); // task1
6:  behavior Task2_body(); // body of task2
7:  behavior Task3_body(); // body of task3
8:
9:  behavior Task2() {
10:    Task1 task1;
11:    Task2_body task2_body;
12:
13:    void main() {
14:       try { task2_body.main(); }
15:       interrupt (e1) { task1.main(); }
16:    }
17: };
18:
19: behavior Task3() {
20:    Task1 task1;
21:    Task2 task2;
22:    Task3_body task3_body;
23:
24:    void main() {
25:       try { task3_body.main(); }
26:       interrupt (e1) { task1.main(); }
27:       interrupt (e2) { task2.main(); }
28:    }
29: };
30:
31: void Main() {
32:    Dummy dummy; // does nothing
33:    Task1 task1;
34:    Task2 task2;
35:    Task3 task3;
36:
37:    void main() {
38:       try { dummy.main(); }
39:       interrupt (e1) { task1.main(); }
40:       interrupt (e2) { task2.main(); }
41:       interrupt (e3) { task3.main(); }
42:    }
43: };
```

Figure 3. Preemption modeling with hierarchically composed exception handling.

3.2 Preemption Modeling with a Run-time Scheduler

In order to solve the two problems mentioned above, we propose another modeling technique for preemptive systems. The key ideas are (a) to model the tasks as ones running in parallel at the top level of the behavioral hierarchy, and (b) to explicitly introduce a run-time scheduler running in parallel with the user-tasks which controls tasks' execution based on their priorities.

Figures 4 and 5 show a SpecC code fragment which models our example system consisting of three user-

tasks with a run-time scheduler[1]. In the *Main* behavior, the scheduler and three user-tasks run in parallel (lines 114–119 in Figure 5). Initially, in the scheduler, all the three tasks are set to be unready (lines 91–93). Then, seven sub-behaviors are executed in parallel (lines 95–103). The first one, *sched_body*, is the main body of the scheduler, i.e., task dispatcher, which sends an event to the highest-priority task to execute it (lines 30–46). Each of the other sub-behaviors (lines 97–102) is dedicated to catching a specific event. For example, *catch_e1* is dedicated to catching event *e1* (lines 61–69). When event *e1* arrives at the system, the exception handler *activate_task1* gets executed (line 77). In *activate_task1*, *task1* is set to be ready to run, then, event *e_sched* is notified to call the task dispatcher *sched_body* (lines 48–53). The dispatcher executes the higher-priority task among ones which are currently ready, i.e., *task1*, by sending event *e1_run* (line 36). Thus, *task1* receives the event and starts its execution (lines 20–22).

In this modeling, preemptions are performed in the following manner. Let us assume that event *e1* arrives during execution of *task2*. Then, *task2* goes to the sleep mode (line 23 of *task2*)[2] and waits for event *e2_run* (line 10). Meantime, *catch_e1* also catches event *e1* (line 67) and *task1* turns ready (line 50). Then, the dispatcher *sched_body* is called to run *task1* (lines 35–36). Thus, *task1* preempts *task2*. Next, let us assume that *task1* finishes its execution. At the end of *task1*, it notifies event *e1_end* to inform the scheduler of its completion (line 25). Then, *catch_e1_end* catches the event (line 77), *task1* turns unready (line 57), and then, the dispatcher *sched_body* is called (line 58). Since *task1* is not ready any longer, the dispatcher notifies event *e2_run* to run *task2* (lines 38–39). Note that *task2* remains ready while it sleeps. Thus, *task2* resumes its execution after completion of *task1*.

In contrast to the previous modeling shown in Figure 3, lower-priority tasks which have arrived during execution of a higher-priority task are not lost. Let us assume that event *e2* is arrived during execution of *task1*. Then, *task1* is suspended (line 23). Meantime, *catch_e2* also catches the event (line 67 of *task2*) and *task2* becomes ready (line 50 of *task2*). Now, both *task1* and *task2* are ready. However, since *task1* has the higher priority, *task1* resumes its execution (lines 35–36). After *task1* is completed, *task2* is started in the same manner described in the previous paragraph.

In this way, the two problems in the previous modeling are now solved.

The example system which we have used up to now consists of only three user-tasks. However, it can be easily seen that our modeling technique is scalable. In case of a system with N user-tasks, we need $(6N + 3)$ behaviors, $(3N + 1)$ events, and N variables of bool type.

[1] In the figures, only one task (*task1*) is described. Note that the other tasks (*task2* and *task3*) are organized in the same way as *task1*.

[2] *task2* is omitted in the figures. For *task2*, please read *task1_sleep* as *task2_sleep*, *e1* as *e2*, and so on.

Table 1. System specification.

	Priority	Period (= deadline)	Execution time
task1	1 (highest)	12 msec.	3 msec.
task2	2 (medium)	17 msec.	4 msec.
task3	3 (lowest)	28 msec.	10 msec.

Table 2. Experimental results.

	Description size (# SpecC lines)	Simulation time
model1	138	18.8 sec.
model2	356	68.3 sec.

4 Preliminary Experiments

We have described a preemptive multi-task system in SpecC based on the two modeling techniques presented in the previous section. The system specification is summarized in Table 1. The system consists of three tasks running concurrently at different execution rates based on a fixed-priority preemptive scheduling policy. Then, we have compiled and simulated the two SpecC programs with SpecC Reference Compiler version 1.0[3].

The experimental results are summarized in Table 2. In the table, *model1* and *model2* denote the SpecC descriptions based on the modeling techniques presented in Sections 3.1 and 3.2, respectively. *model1* consists of 138 lines of SpecC code including comments, while *model2* is of 356 lines. The simulation time is the CPU time when simulation of 1,000,000 cycles (= 1,000,000 msec.) is performed on a Pentium III processor (900 MHz, Linux). *model2* has the overhead of both description size and simulation time mainly due to the run-time scheduler in which more behaviors and events are necessary. In the experiments, no computation is done in the body of each user-task[4]. Therefore, the CPU time is purely the overhead of preemptive scheduling.

[3] SpecC Reference Compiler is available at the web site of UCI [1].

[4] There is only a **waitfor** statement which specifies the amount of delay.

Figure 6 presents the Gantt charts of the simulation results for *model1* and *model2*. In the charts, white boxes (e.g., *task3* from $t = 5$ to 12) denote that the tasks are sleeping because of preemptions by higher-priority tasks. It can be seen that, in *model1*, the second instance of *task2* was lost because *task1* was running when event *e2* happened at $t = 22$. On the other hand, in *model2*, *task2* was executed after completion of *task1* at $t = 23$. At $t = 56$, three events arrived simultaneously. In *model2* all the events were served while two events were abandoned in *model1*.

5 Discussions

Although the second modeling technique presented in Section 3.2 solves the problems mentioned in Section 3.1, it requires many lines of SpecC code. If a preemptive system consists of a large number user-tasks, it would be very tedious and error-prone to describe the system in SpecC. Therefore, it may be a good idea to define a new construct for neatly specifying preemptions on top of the current version of the SpecC language, or to develop a library to support preemptive scheduling.

In the rest of this section, we make another proposal to the SpecC language. In the run-time scheduler of the second modeling (lines 81–105 in Figure 5), we have introduced six sub-behaviors (e.g., *catch_e1*, etc.) in addition to the task dispatcher. The six sub-behaviors have to run in parallel in order not to lose any events in case multiple events are notified simultaneously. Also, for each of the six sub-behavior, an exception handler is necessary (e.g., *activate_task1* for *catch_e1*). This is the reason why the second modeling requires a large number of SpecC lines. This inefficiency can be removed by a small extension of the SpecC exception handling mechanism.

In the current version of the SpecC language, more than one event can be specified for the **interrupt** construct. For example, in the twenty-third line of Figure 4, three events are listed for one exception handler (*task1_sleep*). When at least one event happens, the handler gets executed. Note that the handler cannot know which event of the three has happened. Our proposal is to extend the SpecC exception handling so that exception handlers can somehow know which events have happened even when multiple event have happened at a time. Then, preemptive scheduling can be modeled in a concise manner. Figure 7 shows a SpecC code of the *Sched_body* behavior and the *Scheduler* behavior with the extended exception handling. Compared with the SpecC code in Figures 4 and 5, twelve behaviors (i.e., *catch_e1, catch_e1_end, activate_task1, finish_task1*, etc.) are no longer necessary. Note that, with this modeling in Figure 7, no task is lost when multiple events happen simultaneously.

6 Summary

In this paper we have proposed modeling techniques for fixed-priority preemptive systems in SpecC. We have also made a couple of suggestions to extend the SpecC language, specifically about the SpecC exception handling, so that preemptions can be modeled in a concise manner.

One of our ongoing work is to apply our modeling to several real-world examples. Automated refinement from our modeling in SpecC into RTOS-dependent implementation in the ANSI-C language is also an interesting future work.

References

[1] http://www.cecs.uci.edu/~specc.

[2] *SpecC Technology Open Consortium*, http://www.specc.org.

[3] R. Dömer, A. Gerstlauer, and D. D. Gajski, *SpecC Language Reference Manual Version 1.0*, March 2001.

[4] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.

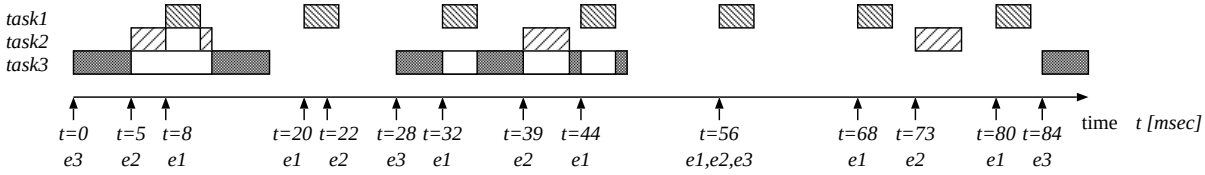
[5] J. Zhu, R. Dömer, and D. D. Gajski, "Syntax and semantics of the SpecC language," In *Proc. of Synthesis and System Integration of Mixed Technologies (SASIMI)*, 1997.

```
1:  event e1, e2, e3; // external signals (e.g., from sensors or timers) to activate user-tasks
2:  event e1_run, e2_run, e3_run; // internal signals from the scheduler to the user-tasks
3:  event e1_end, e2_end, e3_end; // internal signals from the user-tasks to the scheduler
4:  event e_sched; // used for synchronization within the scheduler
5:
6:  bool ready1, ready2, ready3; // true if the corresponding tasks are ready to run
7:
8:  behavior Task1_sleep { // sleep mode
9:     void main() {
10:       wait e1_run; // task1 sleeps until e1_run is notified by the scheduler
11:    }
12: };
13:
14: behavior Task1() {
15:    Task1_body task1_body; // body of task1
16:    Task1_sleep task1_sleep; // sleep mode
17:
18:    void main() {
19:       for (;;) {
20:          wait e1_run; // wait until e1_run is notified by the scheduler
21:
22:          try { task1_body.main(); } // task1 goes into sleep mode when any of the following events happens
23:          interrupt (e1, e2, e3) { task1_sleep.main(); };
24:
25:          notify e1_end; // notify the scheduler that task1 is completed
26:       }
27:    }
28: };
29:
30: behavior Sched_body() { main body of the scheduler (dispatcher)
31:    void main() { // run the highest-priority task among ones which are ready
32:       for (;;) {
33:          wait e_sched; // wait until event e_sched is notified
34:
35:          if (ready1) {
36:             notify e1_run; // execute task1 if it is ready
37:
38:          } else if (ready2) {
39:             notify e2_run; // execute task1 if task1 is not ready and task2 is ready
40:
41:          } else if (ready3) {
42:             notify e3_run; // execute task1 if task1 and task2 are not ready and task3 is ready
43:          }
44:       }
45:    }
46: };
47:
48: behavior Activate_task1() {
49:    void main() {
50:       ready1 = true; // task1 turns ready
51:       notify e_sched; // invoke the scheduler
52:    }
53: };
54:
55: behavior Finish_task1() {
56:    void main() {
57:       ready1 = false; // task1 turns unready
58:       notify e_sched; // invoke the scheduler
59:    }
60: };
```

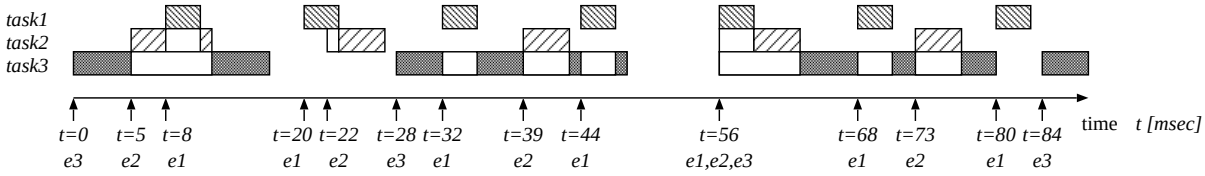
Figure 4. Preemption modeling with a run-time scheduler.

```
61:  behavior Catch_e1() {
62:     Dummy dummy;
63:     Activate_task1 activate_task1;
64:
65:     void main() {
66:        try { dummy.main(); } // do nothing until external event e1 happens
67:        interrupt (e1) { activate_task1.main(); } // activate task1 when e1 happens
68:     }
69:  };
70:
71:  behavior Catch_e1_end() {
72:     Dummy dummy;
73:     Finish_task1 finish_task1;
74:
75:     void main() {
76:        try { dummy.main(); } // do nothing until e1_end happens
77:        interrupt (e1_end) { finish_task1.main(); } // call finish_task1 when task1 is completed
78:     }
79:  };
80:
81:  behavior Scheduler() { // run-time scheduler
82:     Sched_body sched_body;
83:     Catch_e1 catch_e1;
84:     Catch_e2 catch_e2;
85:     Catch_e3 catch_e3;
86:     Catch_e1_end catch_e1_end;
87:     Catch_e2_end catch_e2_end;
88:     Catch_e3_end catch_e3_end;
89:
90:     void main() {
91:        ready1 = false; // initially, all tasks are unready
92:        ready2 = false;
93:        ready3 = false;
94:
95:        par {
96:           sched_body.main();
97:           catch_e1.main();
98:           catch_e2.main();
99:           catch_e3.main();
100:          catch_e1_end.main();
101:          catch_e2_end.main();
102:          catch_e3_end.main();
103:       }
104:    }
105: };
106:
107: behavior Main() {
108:    Scheduler sched; // run-time scheduler
109:    Task1 task1; // task1
110:    Task2 task2; // task2
111:    Task3 task3; // task3
112:
113:    void main(void) {
114:       par {
115:          sched.main(); // the run-time scheduler runs in parallel with user-tasks
116:          task1.main();
117:          task2.main();
118:          task3.main();
119:       }
120:    }
121: };
```

Figure 5. Preemption modeling with a run-time scheduler (cont'd).



(a) model1



(b) model2

Figure 6. Gantt charts of the simulation results.

```
1:  behavior Sched_body() { // main body of the scheduler (dispatcher)
2:     void main() {
3:        if (e1_end is notified) { ready1 = false; }
4:        if (e1_end is notified) { ready2 = false; }
5:        if (e1_end is notified) { ready3 = false; }
6:        if (e1 is notified) { ready1 = true; }
7:        if (e2 is notified) { ready2 = true; }
8:        if (e3 is notified) { ready3 = true; }
9:
10:       if (ready1) { // run the highest-priority task that is ready
11:          notify (e1_run);
12:       } else if (ready2) {
13:          notify (e2_run);
14:       } else if (ready3) {
15:          notify (e3_run);
16:       }
17:    }
18: };
19:
20: behavior Scheduler() {
21:    Sched_body sched_body; // main body of the scheduler (dispatcher)
22:    Dummy dummy; // wait forever
23:
24:    // some event declarations
25:
26:    void main() {
27:       ready1 = false; // initially, all tasks are not ready
28:       ready2 = false;
29:       ready3 = false;
30:
31:       try { dummy.main(); } // wait forever
32:       interrupt (e1, e2, e3, e1_end, e2_end, e3_end) { sched_body.main(); }
33:    }
34: };
```

Figure 7. Preemption modeling with extended exception handling.