Model Checking of OSEK/VDX OS Design Model based on Environment Modeling

Kenro Yatake and Toshiaki Aoki

Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{k-yatake, toshiaki}@jaist.ac.jp

Abstract. This paper presents a model-checking experiment for a design model of a practical real-time operating system (RTOS) based on environment modeling. In previous work, we developed a tool called the environment generator to generate environments for model-checking general RTOS models in Spin. This tool takes a general model of the environments, called the environment model, as an input and generates all possible environments within the bounds of the model. Here, we applied the tool to verify the design model of an OSEK/VDX OS, the RTOS for controlling automotive systems. In this paper, we explain the details of constructing the environment models for verifying various aspects of the RTOS. We also show the results of an experiment using our tool.

1 Introduction

Nowadays, many embedded systems are using real-time operating systems (RTOSs) to realize high functionality and responsiveness. RTOSs are required to have high reliability because they are commonly used in a wide variety of systems. Since failures in embedded systems lead to huge recalling costs, and can even endanger human lives, it is important to ensure the correctness of their behavior. Currently, we are working on a project to verify the design model of an OSEK/VDX OS [8] (hereinafter, the RTOS model) by model checking. OSEK/VDX is the OS specification for controlling automotive systems. The RTOS model that we have developed is described in Promela, the input language of the Spin model checker [6]. We focus on the verification of the task scheduling algorithm. This algorithm is based on priorities, and its correctness is critically important for the safety of automobiles. Specifically, high-priority tasks such as life-saving units must always be activated before all low priority tasks.

To apply model checking to the RTOS model, we need to construct an *environment* that provides function calls to the RTOS model. Such an environment is an application consisting of tasks and resources, and is necessary as a driver to close the behavior of the RTOS model. According to the number of tasks and resources and their priorities, there are a number of environmental variations. For example, one environment may consist of 2 tasks with priorities 1 and 2, whereas another environment consists of 3 tasks with priorities 1, 2 and 3, and a resource with priority 4. Since exhaustively constructing all variations by

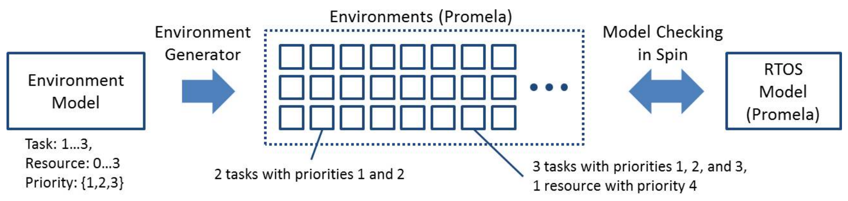


Fig. 1. Environment generator

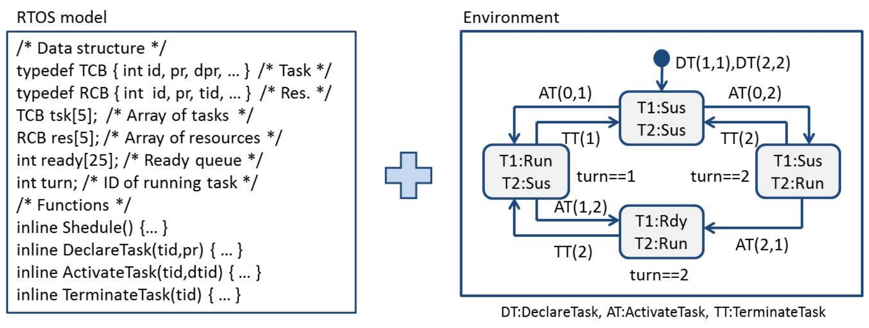
hand is infeasible, we have developed a tool called the *environment generator* to automate the process. Fig. 1 summarizes the concepts behind the tool. The environment generator first takes a general model of the environments, called the *environment model*, as an input. This model consists of class and statechart models, whose notations are based on the unified modeling language (UML) [7], to describe the structure and behavior of the environment. The environment generator then generates all possible environments within the bounds of the model. Finally, each generated environment is model-checked in combination with the RTOS model in Spin.

For the current study, we applied this tool to verify the RTOS model. During verification, we had to construct various environment models to clarify various aspects of the RTOS such as task management, resource management, event control, and interrupt handling. In this paper, we explain the details of constructing the environment models and show experimental verification results.

This paper is organized as follows. In Section 2, we describe the RTOS model and how an environment is used to verify it. In Section 3, we explain the environment model and the environment generator. In Section 4, we explain the verification procedure of the RTOS model by using the environment generator. In Section 5, we discuss the effectiveness of our method. In Section 6, we outline related work. In Section 7, we conclude this paper.

2 RTOS model and its environment

Let us first introduce the RTOS model and explain how it is checked using an environment. The RTOS model is constructed based on the OSEK OS specification, and is described in about 2800 lines of Promela code, following the approach in [2]. The main functionalities of the OS are modeled, including its task management, resource management, event mechanism, and interrupt processing. Fig. 2 (left) shows the basic structure of the code. The first part defines the data structures such as tasks, resources, and the ready queue. Here, tasks are the entities that compete for a central processing unit (CPU), and the currently running task is represented by the variable `turn`. The ready queue is used to manage the execution order of tasks, and is represented by a two-dimensional array (i.e., a set of queues assigned for each task priority level). When a task is



```
/* Data structure */
typedef TCB { int id, pr, dpr, … } /* Task */
typedef RCB { int  id, pr, tid, … } /* Res. */
TCB tsk[5]; /* Array of tasks */
RCB res[5]; /* Array of resources */
int ready[25]; /* Ready queue */
int turn; /* ID of running task */
/* Functions */
inline Shedule() {… }
inline DeclareTask(tid,pr) { … }
inline ActivateTask(tid,dtid) { … }
inline TerminateTask(tid) { … }
```

Fig. 2. RTOS model and an environment

activated, it is pushed onto the queue with corresponding priority. The next task to be executed is then the one at the head of the highest priority queue. The second part of the code defines a number of functions. The function `Schedule` corresponds to the activation of the scheduler, and is called only from inside of the OS. When `Schedule` is called, the next task to be executed is dispatched from the ready queue and the identifier (ID) of the task is stored in `turn`. In contrast, the functions `DeclareTask`, `ActivateTask`, and `TerminateTask` are called from outside the OS. `DeclareTask` registers a new task for the RTOS model. `ActivateTask` and `TerminateTask` activate and terminate a task, respectively, and the main role of these two functions is to enqueue and dequeue tasks for the ready queue. At the end of the execution of these functions, `Schedule` is called to activate the scheduler. Including these three functions, a total of 13 external OS functions are defined. Their executions all follow the same procedure, namely, after conducting their operation, the scheduler is activated.

To apply model checking to the RTOS model, we need an environment that provides function calls to the RTOS model. Fig. 2 (right) shows an example of an environment. In actuality, this environment is implemented in Promela; however, we show it here as a state model for readability. This example environment consists of two tasks: `T1` with priority 1 and `T2` with priority 2. It describes a sequence of function calls to the RTOS model, as well as the state transitions of the tasks expected by the function calls. The environment starts in an initial state constructed by declaring the two tasks using `DeclareTask`. Both tasks are expected to be suspended in this state. When `ActivateTask(0,1)` (activation of `T1` by the OS) is called in this initial state, the environment moves to the left state in which `T1` and `T2` are expected to be running and suspended, respectively. When `ActivateTask(1,2)` (activation of `T2` by `T1`) is called in the left state, the environment moves to the bottom state in which `T1` and `T2` are expected to be ready and running, respectively. `T2` hence preempts the execution of `T1`. In this way, the environment describes the state transitions of tasks expected by the function calls. To ensure that this expectation is actually satisfied, we check the consistency of the environment and the RTOS model by using assertions. In this example, we check that the task running in the environment is also running in the RTOS model by inspecting the variable `turn` of the RTOS model in each state of the environment. For instance, for the left state in which `T1` is expected to be running, we check that `turn` is equal to the ID of `T1` (= 1). In this way, we ensure the correctness of the RTOS model by applying model checking in combination with an environment.

3 Environment generator

The example in the previous section is so simple that we can construct it even by hand. However, as stated in Section 1, there are many environmental variations considering the number of tasks and resources and their priorities. To automate the construction of environmental variants, we previously developed the environment generator tool. We briefly introduce this tool here. For more details, please refer to our earlier work [17].

Environment models are input models of the environment generator. An environment model consists of a class model and statechart models. In the class model, we describe structural variations of the environment. In the statechart models, we describe the state transitions of environment objects, which are expected by the function calls to the target system. Fig. 3 shows an example environment model. The class model defines the following two classes. The first class `RTOS` represents the verification target and defines two types of functions: trigger functions (defined in the upper box of the figure) and reference functions (defined in the lower box). Trigger functions cause state transitions to the environment, and their arguments are defined with a range that characterizes the call variations. Reference functions refer to the internal values of the target system and are used to define assertions. The second class `Task` represents the environment of the target system. It is defined in terms of attributes and associations. For example, the attribute `pr` represents the priority of a task, and the association from `RTOS` to `Task` represents the number of tasks. Such attributes and associations are also defined with ranges that characterize the value and multiplicity variations, respectively. The detailed structure of the environments is constrained by invariants. For example, the invariant of `Task` forces the priorities of tasks to be different from one another. The properties under examination are defined by assertions, which are checked in all states of each object. For example, the assertion of `Task` checks the consistency of the running task between the environment and the RTOS model (as explained in Section 2). The function `GetTurn` is used to reference the value of `turn` in the RTOS model. Invariants and assertions are described in object constraint language (OCL) [16].

The figure also shows the statechart model of `Task`. A transition is caused by calling a trigger function, for example, the transition `AT1` is caused by `ActivateTask` such that when a suspended task is activated, this task runs if no other tasks are running. The expression in the square brackets (`[]`) beneath `AT1` is a guard condition. Simultaneous transitions among multiple objects are defined by *syn-*

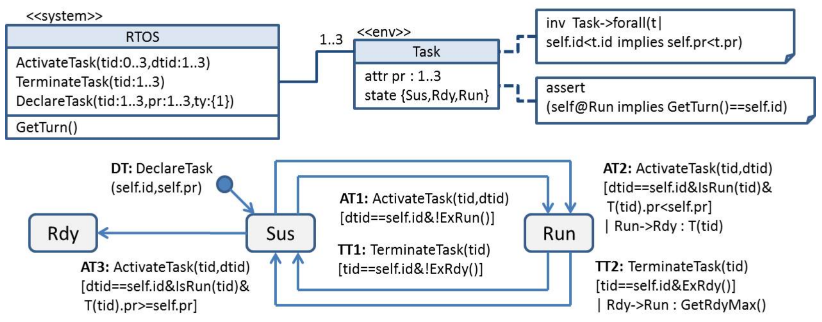


Fig. 3. Example environment model

chronous transitions. For example, the transition `AT2` defines a synchronous transition `|Run->Rdy:GetRun()`. During the main transition, the task transits from suspended to running. In combination with this transition, the task expressed by `GetRun()` (i.e., the running task `GetRun()=Task->select(t|t@Run)`) transits from running to ready. Guard conditions and the objects in synchronous transitions are also described in OCL.

The environment generator generates all possible environments of the environment model. It is implemented as a LINUX command line tool that takes an environment model as a text file input and outputs environments as Promela files. Environment generation is conducted in three steps. Firstly, all possible object graphs are generated within the bounds of the class model. In [17], we performed this generation by using an elementary algorithm that enumerated all of the object graphs in alphabetical order. However, we have now updated the generator to use a satisfiability modulo theories (SMT) solver Yices. The problem of finding object graphs that satisfy the invariants can be considered as a constraint solving problem against the class diagram [3, 4]. We follow this approach, and enumerate all of the object graphs efficiently using Yices [1]. Secondly, for each object graph, we generate a labeled transition system (LTS) by composing the statechart models of all objects in the object graph. Finally, we generate the environments by translating each LTS into a Promela file.

4 Verification of RTOS model

4.1 Approach

To verify the RTOS model, we constructed environment models based on the OSEK/VDX OS specification. For this construction, we followed two approaches: (1) separation of environment models and (2) use case analysis of state transitions.

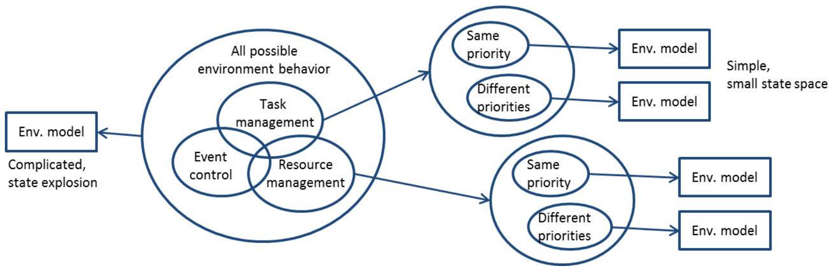


Fig. 4. Separation of environment (Env.) models

Separation of environment models We first constructed separate environment models based on the individual functionality of the OS. Fig. 4 summarizes this concept. The OS has various functionalities such as task management, resource management, event control, and interrupt processing, and we constructed multiple environment models to independently check each of these. There are two reasons for separating the models. Firstly, each environment model is simplified. In actuality, it is possible to construct a single environment model that covers all aspects of the OS. However, we avoided this because the environment model becomes so complicated that the reliability of the verification is reduced. Since the environment model describes the OS properties, it must be simple so as to clearly reflect the intention of the verification. However, if all aspects of the OS are placed into a single environment model, the complexity of the model becomes equivalent to that of the RTOS model. Creating such a model for verification makes little sense because the correctness of the environment model becomes as uncertain as that of the RTOS model. Therefore, we constructed the environment models separately such that each model was as simple as possible. Secondly, the models are separated to reduce the risk of state explosion. If we check all aspects of the OS at once, state explosion can easily occur. However, by separating the environment models, we can check each of them within a relatively small state space. As a result, we can check the entire environment without causing state explosion.

In addition to separation according to functionality, we separated the environment models based on the priority of tasks. Specifically, for each functionality, we constructed two environment models: one consisting of tasks with different priorities and the other consisting of tasks with the same priority. This separation also retains the simplicity of the environment models. To deal with tasks having arbitrary priorities, we would need to define a data structure similar to a ready queue that precisely defines the order of task executions. (For example, let us consider the case where `T1` with priority 2 is running and `T2` and `T3`, both with priority 1, are ready. When `T1` terminates, we cannot tell which of `T2` and `T3` will run next unless we record their activation order.) However, this again increases the complexity of the environment model such that it is equivalent to that of the RTOS model because the RTOS model uses the ready queue. Thus,

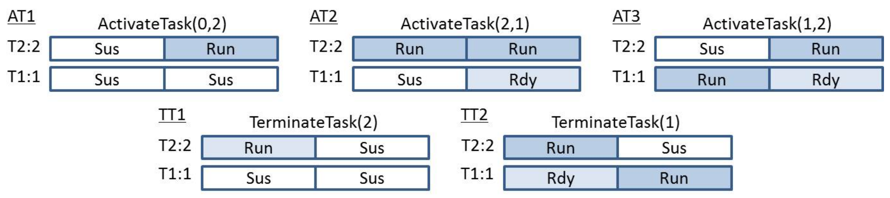
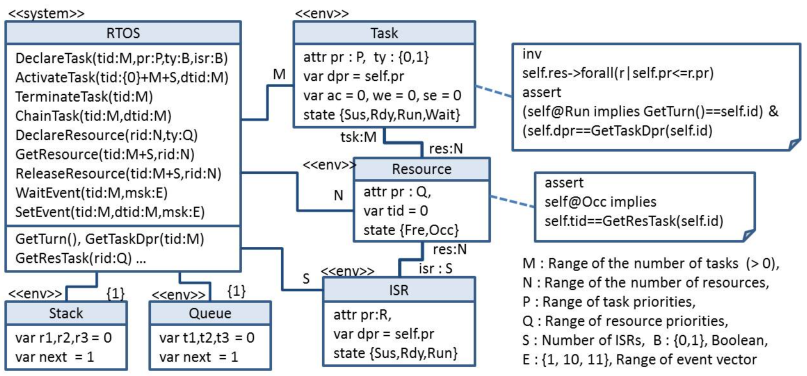


Fig. 5. Use cases for ActivateTask and TerminateTask

we divided the environment models into above two cases. In the different priority case, we can check the preemptive mechanism of task execution, namely, that a task with higher priority is executed before those with lower priorities. Here, we do not need a ready queue because the order of task execution is simply determined by comparing their priorities. In the same priority case, we can check the first-in-first-out order of task execution, that is, tasks with the same priority are executed in activation order. Here, we still need a ready queue to record the order of task activation. However, a single array is adequate to represent the ready queue because there is only one level of priority. This simplifies the environment model compared with the RTOS model, where the ready queue is represented by a two dimensional array.

Use case analysis of state transitions Second, before constructing the environment models, we constructed use case models to understand the specification correctly. In actuality, the behavior of functions is described in various ways in the specification such as by plain text, diagrams, tables, and so on. This variety makes direct conversion of the specification into environment models prone to errors. Hence, we introduced use case models as intermediate models to fill the notational gaps between the specification and environment models. Fig. 5 shows an example of use case models. In such models, we describe the state transition patterns caused by function calls by using concrete objects in the environment. For example, `AT3` presents the state transitions of two tasks, `T1` with priority 1 and `T2` with priority 2, caused by `ActivateTask(1,2)` (activation of `T2` by `T1`). Initially, `T1` is running and `T2` is suspended. After the function call, `T1` then becomes ready and `T2` becomes running. When constructing use case models, it is important to exhaustively cover all state transition patterns. The five cases in the figure are all of the state transition patterns caused by `ActivateTask` and `TerminateTask` in which the tasks have different priorities. Specifically, the three cases of `ActivateTask` are divided into `AT1`, and {`AT2`, `AT3`} based on the existence of a running task. Furthermore, `AT2` and `AT3` are divided based on the relation between the priorities of the running and activated tasks, namely, `AT2` with `T1.pr<T2.pr` and `AT3` with `T1.pr>T2.pr`. The two cases of `TerminateTask` are divided based on the existence of a ready task. Through such a case analysis, we exhaustively extract all state transition patterns from the specification.



<<system>>
RTOS
DeclareTask(tid:M,pr:P,ty:B,isr:B)
ActivateTask(tid:{0}+M+S,dtid:M)
TerminateTask(tid:M)
ChainTask(tid:M,dtid:M)
DeclareResource(rid:N,ty:Q)
GetResource(tid:M+S,rid:N)
ReleaseResource(tid:M+S,rid:N)
WaitEvent(tid:M,msk:E)
SetEvent(tid:M,dtid:M,msk:E)
GetTurn(), GetTaskDpr(tid:M)
GetResTask(rid:Q) ...
M
<<env>>
Task
attr pr : P, ty : {0,1}
var dpr = self.pr
var ac = 0, we = 0, se = 0
state {Sus,Rdy,Run,Wait}
inv
self.res->forall(r|self.pr<=r.pr)
assert
(self@Run implies GetTurn()==self.id) &
(self.dpr==GetTaskDpr(self.id)
tsk:M
res:N
<<env>>
N
Resource
attr pr : Q,
var tid = 0
state {Fre,Occ}
assert
self@Occ implies
self.tid==GetResTask(self.id)
res:N
isr : S
S
<<env>>
ISR
attr pr:R,
var dpr = self.pr
state {Sus,Rdy,Run}
<<env>>
{1}
Stack
var r1,r2,r3 = 0
var next = 1
<<env>>
{1}
Queue
var t1,t2,t3 = 0
var next = 1
M : Range of the number of tasks (> 0),
N : Range of the number of resources,
P : Range of task priorities,
Q : Range of resource priorities,
S : Number of ISRs, B : {0,1}, Boolean,
E : {1, 10, 11}, Range of event vector

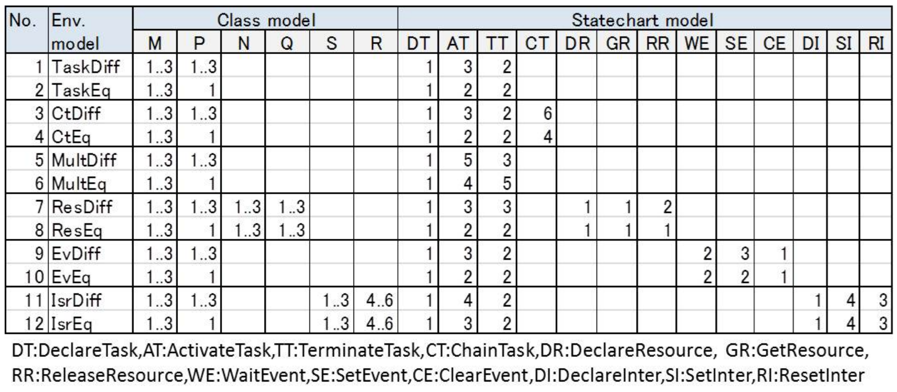
Fig. 6. Class model

We constructed environment models by generalizing the use case models using OCL expressions. For instance, the five cases in the example in Fig. 5 are generalized as the statechart model shown in Fig. 3. The initial conditions are defined by guard conditions and the state transitions of multiple objects are defined by synchronous transitions. Although we have not developed a formal way of analyzing and generalizing use case models, we consider that the case analysis employed to construct the use case models is an important activity for correctly ascertaining the behavior of functions.

4.2 Environment models

Following the approach outlined in the Section 4.1, we defined a total of 12 environment models to check each aspect of the RTOS model. The class model commonly defined for these environment models, is shown in Fig. 6. The ranges of the parameters, invariants, and statechart models are defined individually for each environment model.

Fig. 7 shows the configuration of each environment model. The environment models are divided into six groups based on which the functionalities of the OS are being checked. Each group is further divided into two cases based on the equality of task priorities. For example, environment models No. 1 (TaskDiff) and No. 2 (TaskEq) check the task management functions. They represent the cases with different priorities and the same priority, respectively. The table lists the parameter ranges in the class model. For example, in model No.1, we define the range of task priorities (`P`) as `1..3` and add an invariant to specify that the priorities are different for each task. On the other hand, in model No. 2, we simply define the range as `{1}` to make the priority equal for all tasks. The table also lists the number of transitions in the statechart model. For example, the statechart model of model No. 1, which we have already shown in Fig. 3, consists of six transitions: 1 for `DeclareTask`, 3 for `ActivateTask`, and 2 for



No.	Env.	Class model						Statechart model												
	model	M	P	N	Q	S	R	DT	AT	TT	CT	DR	GR	RR	WE	SE	CE	DI	SI	RI
1	TaskDiff	1..3	1..3					1	3	2										
2	TaskEq	1..3	1					1	2	2										
3	CtDiff	1..3	1..3					1	3	2	6									
4	CtEq	1..3	1					1	2	2	4									
5	MultDiff	1..3	1..3					1	5	3										
6	MultEq	1..3	1					1	4	5										
7	ResDiff	1..3	1..3	1..3	1..3			1	3	3		1	1	2						
8	ResEq	1..3	1	1..3	1..3			1	2	2		1	1	1						
9	EvDiff	1..3	1..3					1	3	2					2	3	1			
10	EvEq	1..3	1					1	2	2					2	2	1			
11	IsrDiff	1..3	1..3			1..3	4..6	1	4	2								1	4	3
12	IsrEq	1..3	1			1..3	4..6	1	3	2								1	4	3

DT:DeclareTask,AT:ActivateTask,TT:TerminateTask,CT:ChainTask,DR:DeclareResource, GR:GetResource, RR:ReleaseResource,WE:WaitEvent,SE:SetEvent,CE:ClearEvent,DI:DeclareInter,SI:SetInter,RI:ResetInter

Fig. 7. Configuration of environment models

`TerminateTask`. As we can see, these functions are used commonly by all of the other models. This is because tasks are necessary to compose the minimum behavior of the environment. We thus treated models No. 1 and No. 2 as core environment models and defined all of the other models by extending these core models. For example, Fig. 8 shows the statechart models of models No. 5, 7, and 9, which are defined by extending model No.1. These three models check the functions associated with multiple activation of tasks, resource management, and event control, respectively. Model No. 5 adds the state transitions `AT4` and `AT5`, which occur when a ready task or a running task is activated multiple times, respectively. Model No. 7 adds the statechart model of the resource, and state transitions `TT1` and `TT2` are updated to include a guard condition that prohibits a task from terminating while occupying resources. Model No. 9 adds the state `Wait` for waiting events. In this way, we constructed each environment model by adding the relevant behaviors to the core models.

Let us explain more about the class model. The attribute `ty` of `Task` represents two types of tasks (basic tasks and extended tasks) defined in the OS. The accessibility of a task to functions is defined based on this type. For example, `WaitEvent` is called only by extended tasks. We check this accessibility in the guard condition of state transitions by referring the attribute. The variable `dpr` of `Task` represents the dynamic priority of a task. Under the priority ceiling protocol of the OSEK/VDX OS, all resources are defined with a ceiling priority (the attribute `pr` or `Resource`). When a task acquires a resource, its priority is changed to the ceiling priority of the resource. In the class model, such dynamic data is defined as a variable, and its value can be changed by actions that are defined along with the state transitions. The invariant of `Task` defines the constraint on the association between `Task` and `Resource` whereby a task can only access resources whose priorities are greater than or equal to that of the task. The classes `Queue` and `Stack` are utility functions for defining the

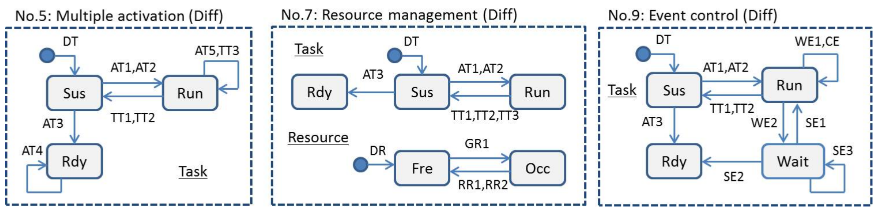


Fig. 8. Statechart models of models No. 5, 7, and 9

precise behavior of tasks. `Queue` is the single ready queue that we explained in Section 4.1, and is used to record the activation order of tasks for models with the same priority. `Stack` is used to control the last-in-first-out order of resource accessing, specifically, when a task releases a resource, that resource must be the last one acquired by the task. The class `ISR` represents the interrupt service routines. ISRs are those entities besides tasks that compete for the CPU. Interrupt processing is checked in models No. 11 and12, which use the functions `DeclareInterrupt`, `SetInterrupt` and `ResetInterrupt`. These functions are not defined in the specification; however, we included them in the RTOS model to represent the activation and termination of ISRs. Interruption is then realized by calling these functions from the environment model. Since the processing level of ISRs is higher than that of tasks, the range of the ISR priorities (`R`) is set to be higher than that of the task priorities. For the assertions, we checked the consistency of running tasks between the RTOS model and the environment for all models. We also checked the consistency of the dynamic priority of tasks and the consistency of the owner task of a resource in the resource management models (No. 7 and 8).

4.3 Verification results

Fig. 9 shows the results of environment generation and model checking. The environment generation results show the number of environments generated from each environment model, the time taken for generation, the average length of the Promela files, and the average number of states and transitions contained in each environment. The model checking results show the time taken for checking all of the environments and the number of environments in which errors were detected. From Fig. 7, we limited the number of tasks, resources, and ISRs to a maximum of 3. With these ranges, we were able to generate a total of 620 environments in about 1 min, which is quite efficient since only about 0.1 s was needed to generate each environment. This result demonstrates the effectiveness of using the SMT solver. For model checking, we were able to check the RTOS model using all of the environments without state explosion occurring due to the separation of the environment models. The entire model checking took 106 min such that about 10 s per environment was required on average. Most of this time was used for compilation, which grows exponentially with the length of the

No.	Env.	Environment Generation					Model Check	
	model	Num	Time(s)	Lines	States	Trans	Time(s)	Errors
1	TaskDiff	26	0.5	153	4	9	113.8	0
2	TaskEq	14	0.4	273	10	19	70.7	0
3	CtDiff	26	0.6	183	4	17	116.3	19
4	CtEq	14	0.5	343	10	37	76.4	13
5	MultDiff	26	0.7	199	8	19	119.1	0
6	MultEq	14	1.6	597	50	99	106.3	0
7	ResDiff	248	38.3	878	44	110	2904.0	188
8	ResEq	98	15.4	1079	53	136	1702.0	56
9	EvDiff	26	1.2	336	15	47	143.0	0
10	EvEq	14	2.3	1271	78	251	253.6	0
11	IsrDiff	182	10.3	502	17	49	1249.7	0
12	IsrEq	98	9.6	903	40	117	1964.4	0

No.7			
Task	Res	Num	Errors
1	1	8	0
1	2	22	14
1	3	62	54
2	1	9	0
2	2	27	18
2	3	81	72
3	1	3	0
3	2	9	6
3	3	27	24

Fig. 9. Verification results (CPU: 2.4 GHz, Memory: 4.0 GB)

Promela file. The cost of model checking can therefore be improved by using PC clusters, that is, checking all of the environments in parallel.

The results show that errors were found in the environments of models No. 3, 4, 7, and 8. The RTOS model and the environment models were constructed by different researchers. Hence, when an error was discovered, we needed to identify which model had caused it. The results show both situations. For models No. 3 and 4, the errors were caused by the environment models. Specifically, the errors were contained in the state transition of `ChainTask`. When `ChainTask(t1,t2)` is called, it first terminates task `t1` and then activates task `t2`. If task `t2` is already running or is ready, however, then the task is activated twice. Such multiple activation is allowed only for basic tasks. Nevertheless, the environment models did not check this condition and allowed multiple activation also for extended tasks. As a result, the RTOS model flagged an error to notify the violation of this condition. For models No. 7 and 8, the errors were caused by the RTOS model. Under the priority ceiling protocol, the priority of the task is changed only when it acquires a resource whose priority is higher than that of the task. However, the RTOS model was defined to change the priority every time a task acquired a resource. As a result, the assertion in the environment was violated due to the inconsistency between the dynamic priority of the tasks of the RTOS model and those of the environment. In both of the above cases, when errors were found, we held a discussion to identify the cause of the errors. This activity allowed us to understand the specification correctly, and led us to construct an error-free RTOS model within the bounds of the environment models.

The results also show a feature of our method: *structural difference analysis of errors*. The table on the right-hand side of Fig. 9 lists the number of errors found in the environments of model No. 7 grouped according to the number of tasks and resources. For example, among the 27 environments containing 2 tasks and 2 resources, 18 environments caused errors. From this table, we can see that errors were caused by environments containing more than one resource. This result leads us to infer that an error occurs when a task tries to acquire multiple resources, and we further infer that changes in the dynamic priority are not performed correctly. From structural information such as this, we can predict the cause of errors before analyzing counterexample traces.

5 Discussion

5.1 Effectiveness

In the environment model, we can declare the structural variation of the environment in OCL using the ranges of the parameter and invariants. From this model, we can then exhaustively generate all environmental variants using the environment generator within the given bounds. As shown in the experiment, the number of environments can become so large that it is almost impossible to correctly construct them all by hand. However, the environment generator enables automatic and effective generation of environments with leveraging the SMT solver. The generator can also exhaustively enumerate all structural variations, which is important for obtaining a full coverage of verification with respect to all environment structures. Thanks to this exhaustiveness, we were able to find an error in the resource manipulation without missing the case in which a task is linked to multiple resources.

Our method is also effective for avoiding state explosion. The act of generating all structural variations of the environment is equivalent to structurally decomposing the entire environment into smaller environments. This enables us to check each environment independently within a small state space. As a result, we can check the entire environment without causing state explosion. We further benefit from this decomposition since it facilitates structural difference analysis of errors. The effect of the decomposition is thus notable, especially when it is applied to the verification of systems that have various environmental structures. In this sense, our method is most effective when applied to the domains of OSs and middleware.

5.2 Verification Coverage

In our method, the verification coverage must be evaluated with respect to the structure and behavior of the environment. For the structural coverage, we limited the numbers of tasks, resources, and ISRs to a maximum of 3. We consider that this number is sufficient to check the critical properties of the OS. For example, to check that the priority ceiling protocol is satisfied, we need to examine the following properties. (1) When a task occupies multiple resources, its dynamic priority is set to the maximum priority of the resources. (2) A task with higher dynamic priority must be executed before that with lower dynamic priority. To check (1), we need to create a situation in which a task can access at least two resources. To check (2), we need to create a situation in which two tasks occupy at least one resource. Therefore, we need at least 2 tasks and 2 resources and must define the ranges accordingly to cover these numbers. Basically, the sufficient ranges depend on the system under verification. Thus, when defining the ranges, it is important to identify which properties to check and the structure that needs to be created to check them. The ranges must initially be defined to at least cover that structure. Then, they should be extended further to raise the structural verification coverage depending on the machine power. Currently, the environment generator efficiently generates environments for up to 8 objects on a PC with average specifications. For a greater number of objects, the composition of the statechart models becomes the bottleneck and generation takes hours.

For the behavioral coverage, we have so far defined statechart models for checking normal execution sequences of the OS. However, for such execution sequences, we still need to check the interaction between functionalities. For example, we need to check the case where ISRs can access resources and set events. When we check many functionalities simultaneously, we risk complicating the statechart models. We consider that the combination of 2 functionalities is a limitation for maintaining the simplicity of the model. For checking further combinations, a simple and specific environment should be constructed directly by hand on the Promela level by abondoning the generarity of the environment model. For abnormal execution sequences, we can construct statechart models for verifying them by extending those of normal execution sequences. For example, when `ActivateTask` is called for a running task, an error must be flagged by the OS. This can be checked by adding a transition from the running state to an error state that is entered when `ActivateTask` is called for a running task. In the error state, we check that the error is actually flagged in the RTOS model by referring the variable representing the error code.

Moreover, we need to strengthen the verification of interrupt processing. We modeled the activation and termination of ISRs by calling the functions `SetInter` and `ResetInter` from the environment. However, we only call them before and after other functions calls, and we have not checked the case where they are called *within* a function execution. To enable such a check, we need to define ISRs as different processes from the process of the environment. This then allows ISRs to interleave the functions of the environment. Still, the difficulty remains of how to realize the nested activation of ISRs and the function call from ISRs. Furthermore, allowing interleaving at any point of a function execution can increase the risk of state explosion. Considering these problems, we need to develop an effective way to verify interrupt processing.

6 Related work

Tkachuk et al. [15] proposed the Bandera Environment Generator (BEG) to automatically generate environments for the verification of Java programs in Bandera. BEG has been used to verify commercial software and a web application [14, 12]. BEG generates environments from specifications written by the user, called environment assumptions, or by analyzing the programs that implement the environment. The environment assumptions are described through regular expressions as sequences of method calls. Their approach corresponds to describing a single instance of the environment model in our method; however, by expressing the set of instances as a class model, we can automatically generate all possible instances based on variations of this model.

Penix et al. [11] validated the time partitioning of DEOS RTOS by using Spin. In their method, environments are obtained by filtering a nondeterministic environment with linear temporal logic (LTL) assumptions [10]. The method uses a top-down approach, where an over-approximated environment is gradually reduced by LTL assumptions toward the ideal environment. In comparison, our approach is bottom-up, that is, we start with an under-approximated environment and gradually extend it toward the ideal environment. Our method enables execution of this process by using statechart models, which are a familiar concept for general software engineers. In this sense, we consider that our method is more effective in practical settings.

Dhaussy et al. [5] proposed a formal language called context description language (CDL) for describing system environments. In their method, environment behavior is defined by using actors and sequence diagrams. Through CDL, a set of traces representing the interaction between the environment and the system are generated. Each trace is checked in combination with the system, and the property automata are examined by model checking. Raji et al. [13] extended Dhaussy et al.fs method by introducing use case diagrams to facilitate the description of an environment. Similar to those works, we also make use of UML to describe environments. However, we address the problem of structural variation by using a class model, which is crucial for the verification of OSs.

Parizek et al. [9] proposed a method that combines the Java PathFinder and Protocol Checker model checkers. Their method targets the validation of Java components whose protocols are described in architecture description language (ADL). Model checking is conducted by searching the program states using Java PathFinder in the Java parts and Protocol Checker in the ADL parts. Although the environment in their method can be modeled in ADL, environmental variation cannot be expressed using classes as in our method.

7 Conclusion

In this paper, we presented a model-checking experiment for an OSEK/VDX OS design model based on environment modeling. In the environment model, we define the structure and behavior of the environment using class and statechart models. From this environment model, the environment generator generates all possible environmental variations within the bounds of the model. To verify various individual aspects of the RTOS model, we constructed separate environment models. By this separation, each model was simplified and the risk of state explosion was avoided. By using the environment generator, we were able to efficiently generate a sufficient range of environments to verify the critical properties of the RTOS model. By checking the RTOS model using the generated environments, we have shown that the correctness of the RTOS model can be guaranteed within the given experimental bounds. As future work, we need to verify other aspects of the RTOS model including abnormal execution sequences and interrupt processing. We will also consider the development of a formal way to analyze use case models and to extend the environment models.

References

1. Yices: An SMT Solver. http://yices.csl.sri.com/.
2. Toshiaki Aoki. Model Checking Multi-Task Software on Real-Time Operating Systems. In *ISORC*, pages 551–555. IEEE Computer Society, 2008.
3. Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80, Washington, DC, USA, 2008. IEEE Computer Society.
4. Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking Unsatisfiability for OCL Constraints. *ECEASST*, 24, 2009.
5. Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine Raji, Yves Traon, and Benoit Baudry. Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 438–452, Berlin, Heidelberg, 2009. Springer-Verlag.
6. G.J.Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.
7. OMG. Unified Modeling Language. http://www.uml.org/, 1989.
8. OSEK/VDX. OSEK/VDX Operating System Specification 2.2.3. http://portal.osek-vdx.org/, 2005.
9. Pavel Parizek and Frantisek Plasil. Partial Verification of Software Components: Heuristics for Environment Construction. In *EUROMICRO-SEAA*, pages 75–82. IEEE Computer Society, 2007.
10. Corina S. Pasareanu. DEOS Kernel: Environment Modeling using LTL Assumptions. Nasa ames technical report nasa-arc-ic-2000-196, NASA Ames Research Center, 2000.
11. John Penix, Willem Visser, Seungjoon Park, Corina S. Pasareanu, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verifying Time Partitioning in the DEOS Scheduling Kernel. *Formal Methods in System Design*, 26(2):103–135, 2005.
12. Sreeranga P. Rajan, Oksana Tkachuk, Mukul R. Prasad, Indradeep Ghosh, Nitin Goel, and Tadahiro Uehara. WEAVE: WEb Applications Validation Environment. In *ICSE Companion*, pages 101–111. IEEE, 2009.
13. Amine Raji and Philippe Dhaussy. Use Cases Modeling for Scalable Model-Checking. In *APSEC'11*, page xx, Minh City, Viet Nam, December 2011.
14. Oksana Tkachuk and Matthew B. Dwyer. Environment generation for validating event-driven software using model checking. *IET Software*, 4(3):194–209, 2010.
15. Oksana Tkachuk, Matthew B. Dwyer, and Corina S. Pasareanu. Automated Environment Generation for Software Model Checking. In *ASE*, pages 116–129. IEEE Computer Society, 2003.
16. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
17. Kenro Yatake and Toshiaki Aoki. Automatic Generation of Model Checking Scripts Based on Environment Modeling. In Jaco van de Pol and Michael Weber, editors, *SPIN*, volume 6349 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2010.