Automatic Generation of Model Checking Scripts 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. When applying model checking to the design models of the embedded systems, it is necessary to model not only the behavior of the target system but also that of the environment interacting with the system. In this paper, we present a method to model the environment and to automatically generate all possible environments from the model. In our method, we can flexibly model the structural variation of the environment and the sequences of the function calls using a class model and statechart models. We also present a tool to generate Promela scripts of SPIN from the environment model. As a practical experiment, we applied our tool to the verification of an OSEK/VDX RTOS design model.

1 Introduction

Recently, model checking is drawing attention as a technique to improve the reliability of software systems [16]. Especially, they are widely applied to a verification of embedded systems. The major characteristics of embedded systems is reactiveness. i.e., they operate by the stimulus from the environment. For example, Real-Time Operating Systems (RTOS), which are embedded in most of the complex embedded systems, operate by the service calls from the tasks running on them. In order to apply model checking to such systems, it is necessary to model not only the behavior of the target system but also that of the environment.

The most typical approach to model an environment is to construct a process which calls all the functions provided by the system non-deterministically. Although it realizes an exhaustive check for all the possible execution sequences, description of properties tends to become complicated because it needs extra assumptions to filter out uninterested sequences from all the sequences. Furthermore, it is prone to suffer state explosion because all the sequences are checked at a time. Another approach is to call specific sequences of functions depending on the properties to check. For example, we limit the range of the function calls to the normal execution sequences and check that certain properties hold in that range. The advantage of this approach is that the property description becomes simple and precise because the assumptions of the properties are implied by the sequences themselves, Furthermore, as the range is limited, we are more likely to be able to avoid state explosion.

We consider the latter approach is more realistic. However, we must further consider the structural variation of the environment. For example, the environment of an RTOS consists of a multiple number of tasks and resources. There are also various patterns in their priority values and function call relationships. Although we need to check the system for all the environment variations, the number of the variations is so large that we cannot construct them by hand.

To cope with this problem, we propose an environment modeling method where we model the environment variations in a model called *environment model* and automatically generate all possible environments from the model. To allow the use in practice, we defined the environment model based on UML [9]. In a class model, we can model the structural variation of the environment. In statechart models, we can define the sequences of the function calls. Our method is implemented as a tool called *environment generator* which inputs an environment model and outputs environments as Promela scripts of SPIN. As a practical experiment, we applied the tool to the verification of an RTOS design model which is based on OSEK/VDX RTOS specification [1]. In this paper, we explain the details of our method and the verification experiment of the RTOS model.

This paper is organized as follows. In section 2, we explain the approach of our method. In section 3, we explain the environment model. In section 4, we explain the environment generation and the tool. In section 5, we show the verification experiment. In section 6, we discuss the coverage and parallelization of our method. In section 7, we give a conclusion and future work.

2 Approach

Let us explain how reactive systems are verified using an environment. Fig.1 shows an RTOS and its environment. The RTOS implements data structures such as a task control blocks and a ready queue and provides functions such as `ActivateTask()` and `TerminateTask()`. If these functions are called, the RTOS schedules the tasks based on their priorities. To verify this behavior, we prepare an environment, for example, consisting of two tasks T1 and T2 (T2's priority is higher than T1's). This environment describes a sequence of function calls and state transitions of the tasks expected by the calls. For example, if the function `ActivateTask()` is called to T2, T2 is expected to become running. Then, if the same function is called to T1, T1 is expected to become ready. To verify that the RTOS satisfies this expectation, we apply model checking to the RTOS in combination with the environment. Specifically, in each state of the environment, we check the consistency between the environment state and the internal values of the RTOS. For example, if T1 and T2 are ready and running, the ready queue in the RTOS must contain the identifier of T1, and the variable `turn` (representing the running task) must contain the identifier of T2. By checking this consistency, we verify the behavior of the RTOS.

The problem of this approach is that this environment is only one of the cases of the large number of environment variations. We need to verify the RTOS for

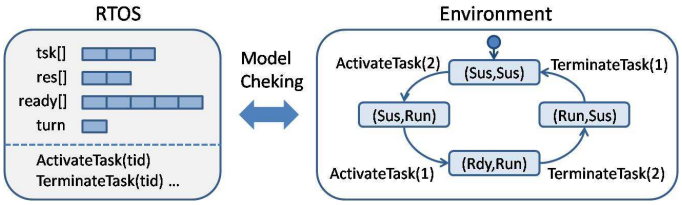

Fig. 1. Model checking with an environment

all the variations with respect to the structures such as the number of tasks and resources, the patterns of the priority values and function call relationships. But it is unrealistic to construct all of them by hand. One could think of constructing a general environment with m tasks and n resources, but this is prone to suffer state explosion.

To cope with this problem, we introduce a model to describe the environment variations and automatically generate all the environments from the model. Fig. 2 summarizes this idea. To verify the target system (RTOS design model), we first construct an environment model. This model is based on UML in which all possible environment structures are defined using parameters with ranges. Then, we generate all the environment instances from the model using the environment generator. All the generated instances, which are structurally different from each other, cover all the variations of the environment. The environment generator inputs an environment model as a text file and outputs the environments as Promela scripts. Finally, we combine the target system with each environment instance and conduct model checking using SPIN.

Our approach has the following advantages:

1. **Easy syntax**: The syntax of the environment model is based on UML which is familiar to most of the software engineers. This lowers the hurdle for introducing our method to the software development in practice.
2. **Alleviation of state explosion problem**: It alleviates the state explosion problem by structurally dividing the whole environment into individual environment. As each environment can be checked in a relatively small state space, we are likely to be able to check the whole environment without causing state explosion.
3. **Difference analysis on debugging**: When a bug is detected in model checking, our method allows us to structurally analyze its source, i.e., we can identify the structural boundary of the source of a bug by comparing the check results. For example, when the two results "The case of 2 tasks and no resources is correct" and "The case of 2 tasks and 1 resource is not correct" are obtained, we can presume that the resource handling function contains a bug.
4. **Generality**: It can be generally applied to the verification of reactive systems. Our method is especially effective for the systems whose environment

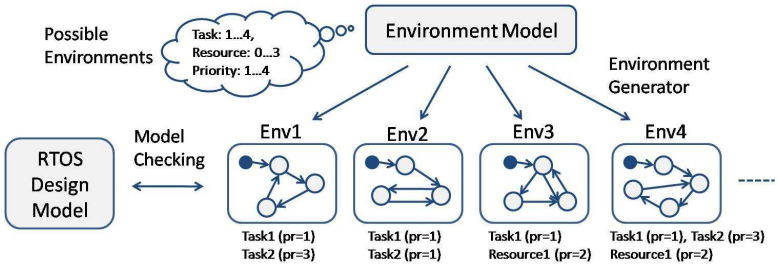

Fig. 2. Environment modeling method

has a lot of structural variations. Examples of such systems are operating systems and middleware systems.

3 Environment models

In this section, we explain environment models with an example of RTOS. We also present formal definitions in A.1.

3.1 Class model

Fig.3 shows the class model of the environment for an RTOS. The class model consists of a class representing a target system and classes representing its environment. In the figure, the class `RTOS` is the target system and the two classes `Task` and `Resource` are the environment classes.

The target class defines two kinds of functions as the interface with the environment. The functions labeled with `fun` are *trigger functions*. They trigger the state transition of the target system. For example, `ActivateTask(tid,dtid)` is the function to activate the task of ID `dtid` (`tid` is the ID of the caller task). The argument of a function is defined with a range like `tid:1..M`. representing the variation of the arguments. The functions labeled with `ref` are *reference functions*. They refer the internal values of the target system. They are used to define assertions (explained later in this section).

The environment classes are defined with attributes, associations, and variables. They are labelled with `attr`, `assoc`, and `var`, respectively. An attribute is defined with a range like `pr:1..P` representing the variation of the attribute values. (`pr` is a priority of a task.) An association is also defined with a range like `res:0..N`. representing the variation of the multiplicity, i.e., the number of objects linked with an object. The associations from the target class to an environment class defines the number of objects which instantiate from the environment class. A variable is a data of an object which can be dynamically changed along with state transitions. It is defined with a default value.

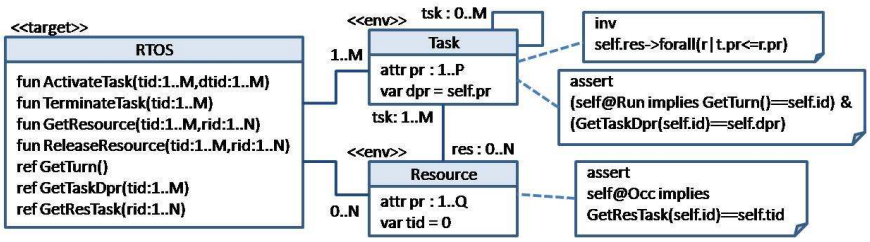

Fig. 3. The class model of an RTOS environment

Invariants can be defined for environment classes. They are written in OCL [20]. An invariant defines a constraint on the structure of objects. For example, `Task` defines an invariant constraining the pattern of links from a task to resources. This invariant reflects the description of the specification that a task can only acquire the resources whose priorities are equal to or higher than that of the task. The OCL expressions in our model is a subset of OCL containing the set operations and the state reference operations.

Assertions are defined for environment classes. An assertion defines a predicate which is checked in each state of objects. In an assertion, the internal values of the target system can be accessed by reference functions to define the consistency between the target system and the environment. For example, `Task` defines an assertion to check if the variable `turn` in the RTOS is equal to the identifier of the running task in the environment. It also checks that the runtime priority of a task is the same in the RTOS and the environment. The reference functions `GetTurn()` and `GetTaskDpr()` are used to obtain the value of `turn` and the runtime priority of the task in the RTOS.

3.2 Statechart models

In statechart models, we define the state transitions of environment objects expected for the function calls of the target system. Fig.4 shows the statechart models of `Task` and `Resource`. They describe the normal execution sequences of RTOS. A transition occurs by the call of a trigger function. For example, the transition (1) is caused by `ActivateTask()`. The expression in `[]` is a guard condition described in OCL. In the model, typical expressions are defined as functions like `ExRun()=Task->exists(t|t@Run)`.

A set of *synchronous transitions* can be attached to a transition. By synchronous transitions, we can define the transitions of other objects which occur synchronously with the transition of the self object. For example, the transition (2) defines the synchronous transition `Run->Rdy : GetRun()`. This means: "Along with the transition of the self object, the `Task` object obtained by the OCL function `GetRun()` (the task which is in the state `Run`) transits from the state `Run` to `Rdy`. Currently, asynchronous transitions are not supported.

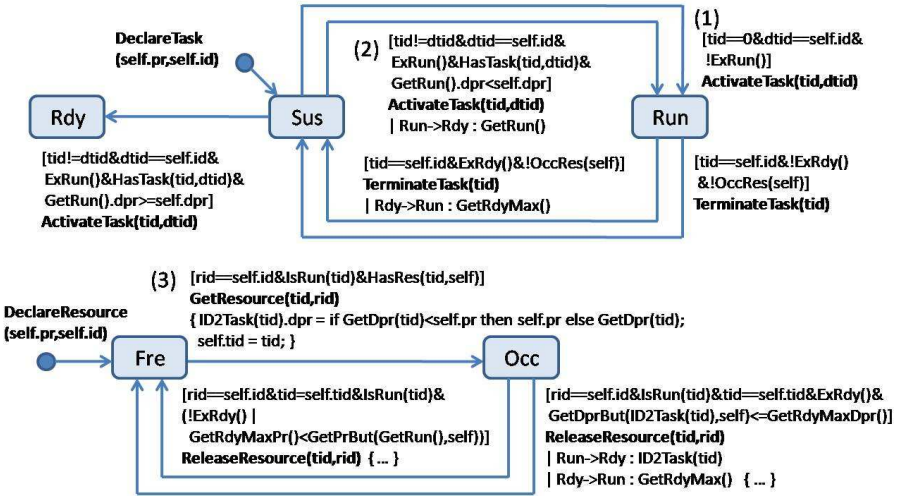

Fig. 4. The statechart model of `Task` (top) and `Resource` (bottom)

An action can be attached to a transition. It is a sequence of statements described in `{}` which are executed along with the transition. In an action, variables of objects are updated. For example, the transition (3) defines an action with two statements which update the variables `dpr` of a task (the runtime priority), and `tid` of a resource (the task ID occupying the resource), respectively.

4 Generation of environments

In this section, we explain the environment generation with the example. As shown in Fig. 5, it is done in three steps: (1) Generation of object graphs, (2) composition of statechart models, and (3) translation into Promela scripts. We also present formal algorithms for (1) and (2) in A.2 and A.3.

4.1 Generation of object graphs

Firstly, we generate all the possible object graphs from the class model. An object graph is represented by a set of objects which instantiate from all the classes. Each object holds the values of attributes and associations. The value of an association is a set of objects with which the object links.

We generate object graphs based on a data structure called *graph counter* which represents an object graph by a vector of natural numbers. By counting up the counter, we enumerate all the variation of object graphs. Let us consider the example in Fig. 3 with `M=2`, `N=1`, `P=2`, and `Q=2`. The graph counter for this model consists of 8 numbers. Each of them corresponds to `T1.pr` (the task `T1`'s attribute `pr`), `T1.tsk`, `T1.res`, `T2.pr`, `T2.tsk`, `T2.res`, `R1.pr` and

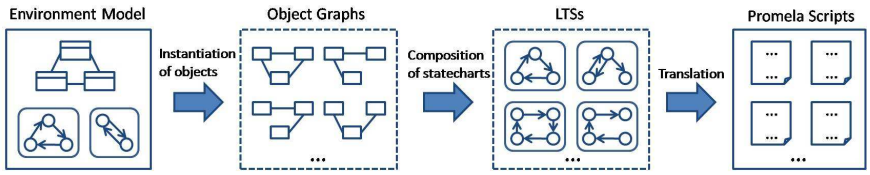

Fig. 5. Environment generation

`R1.tsk`, respectively. For example, the graph counter `L=<2,3,1,1,4,2,2,3>` represents an object graph with `T1.pr=2`, `T1.tsk=[T2]`, `T1.res=[]`, `T2.pr=1`, `T2.tsk=[T1,T2]`, `T2.res=[R1]`, `R1.pr=2`, and `R1.tsk=[T1,T2]`. `L[1]=2` means that the `T1.pr`'s value is the 2nd value of the `pr`'s domain `{1,2}`. Likewise, `L[2]=3` means that the `T1.tsk`'s value is the 3rd value of the `tsk`'s domain `{[],[T1],[T2],[T1,T2]}`.

To enumerate all the graphs, we start with the initial counter `<1,...,1>` and count it up repeatedly to `<1,...,1,2>`, and then, `<1,...,1,3>`. If `L[8]` reached the maximum number `3` (`R1.tsk`'s domain is `{[1],[2],[1,2]}`), the next count causes carry over. So, the next counter is `{1,...,2,1}`. We repeat counting until the counter reaches the maximum value `<2,4,2,2,4,2,2,3>`. Along with the counting, we translate the graph counter into an object graph and output it if it satisfies invariants.

The computation time of this algorithm increases exponentially as the multiplicities of associations increase. However, the computation space is limited to $O((A+B)\times C)$. (A,B,C is the number of attributes, associations, and objects.)

4.2 Composition of statechart models

Next, we compose the statechart models of all objects in each object graph. The result of the composition is an LTS (Labelled Transition System) [8]. Fig.6 shows an example of an object graph and its LTS. In the LTS, each state contains the states and variables of all the objects. For example, in the initial state (A), the tasks T1 and T2 are in the state `Sus` and the resource R1 is in the state `Fre`. The variables `dpr` of T1 and T2 are `1` and `3`, respectively. The variable `tid` of R1 is `0`.

Transitions in statechart models are added to the LTS if their guard conditions are evaluated to true in the LTS state and the object graph. For example, the transition from (A) to (B) by the function `ActivateTask(0,1)` (`AT(0,1)`) is added because the guard condition of the transition (1) in Fig. 4 becomes true for the object T1 and the action argument `(0,1)` in the state (A). If a transition has synchronous transitions, synchronized objects are obtained by evaluating the OCL expression. The synchronized objects transit along with the transition of the self object. For example, in the transition from (C) to (D) by the function `ReleaseResource(1,1)` (`RR(1,1)`), along with the self object R1 transits from `Occ` to `Fre`, the synchronized objects T1 and T2 transit from `Run` and `Rdy` to

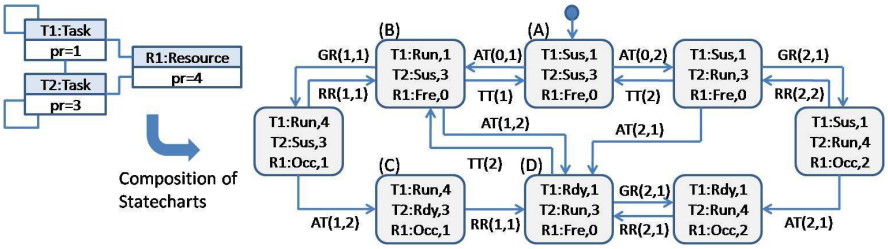

Fig. 6. An object graph and its LTS

`Rdy` and `Run`, respectively. This transition also updates the variables of T1 and R1 by executing the corresponding action.

It could be possible to translate all the statechart models directly into Promela and leave the composition to SPIN. However, we do not do so because the OCL expressions attached with the transitions are hard to express directly in Promela. So, we evaluate the OCL expressions and compose the statechart models at this point.

4.3 Translation into Promela scripts

Finally, we translate each LTS into Promera script. A state of LTS contains the states and variables of all objects. In Promela, the states of objects are represented by a label and the variables of objects are represented by the variables of Promela. For example, Fig. 7 shows the Promela script corresponding to the state (D) of the LTS in Fig.6. In the script, the three states `Rdy`, `Run`, and `Fre` of the three objects `T1`, `T2`, and `R1` are represented by the label `Rdy_Run_Fre`. The variables of the three objects are represented by the variables `Task1.dpr`, `Task2.dpr`, and `Resource1.tid`. These variables are checked in the guard conditions of the if-statement and updated appropriately after calling the functions `TerminateTask(2)` and `GetResource(2,1)`.

The assertion is checked at the beginning of the state. It is a conjunction of the assertions of all the objects. (In this case, only the assertion of two tasks are conjuncted as the assertion of the resource is true.) The three variables `ret_GetTurn`, `ret_GetTaskDpr_1`, and `ret_GetTaskDpr_2` represent the internal values of the RTOS which are obtained by the reference functions `GetTurn()`, `GetTaskState(1)`, `GetTaskState(2)`, respectively. These variables are set to the return values of the corresponding functions in the inline function `set_ref()` before the assertion check. Expressions in the assertions except for these variables are evaluated during the composition of statechart models.

4.4 Environment generator

We implemented the environment generator as a command line tool which inputs an environment model as a text file and outputs all the environments as Promela

```
Rdy_Run_Fre:
  set_ref();
  assert ((ret_GetTaskDpr_1==1) &&                  /* Task1 */
          (ret_GetTurn==2 && ret_GetTaskDpr_2==3)); /* Task2 */
  if
  :: Task1.dpr==1 && Task2.dpr==3 && Resource1.tid==0 -> TerminateTask(2);
     Task1_var.dpr=1; Task2.dpr=3; Resource1.tid=0; goto Run_Sus_Fre;
  :: Task1.dpr==1 && Task2.dpr==3 && Resource1.tid==0 -> GetResource(2,1);
     Task1.dpr=1; Task2.dpr=4; Resource1.tid=2; goto Rdy_Run_Occ;
  fi;
```

Fig. 7. Promela script corresponding the state (D) in Fig. 6

files. This tool can be applied generally to any systems as long as we observe the interface between the environment model and the target system. (The Promela script of the target system must contain the inline functions corresponding to the trigger functions and reference functions in the environment model.) Fig. 8 shows the architecture of the tool. It mainly consists of three components realizing the three steps of the environment generation: the graph generator, the state composer, and the Promela translator. It also has the invariant filter for checking invariants for object graphs, and the OCL evaluator for evaluating OCL expressions.

5 Experiment

As an application of our method to a practical system, we conducted an experiment to verify that an RTOS design model conforms to the OSEK/VDX RTOS specification. The design model is implemented in Promela (about 1800 lines) following the approach in [2]. For the verification, we constructed the environment model based on the specification. We have presented this model partially in Fig. 3 and Fig. 4. We generated the environments by the environment generator and conducted model checking for some of them in SPIN.

5.1 Environment generation

Table. 1 shows the number of generated environments from the model in Fig. 3 with the constants `M`, `N`, `P`, and `Q` set to 4, i.e., tasks and resources are created up to 4 and both of them can take priorities up to 4. We generated environments by moving the number of tasks and resources from 1 to 4. In the Table. 1, T is the number of tasks and R is the number of resources. Actually, the specification does not limit the number of tasks and resources and it allows the number of priorities up to 16. In our model, however, we limited the maximum number of objects and priorities to 4 so that the environment generation can be completed within a reasonable time.

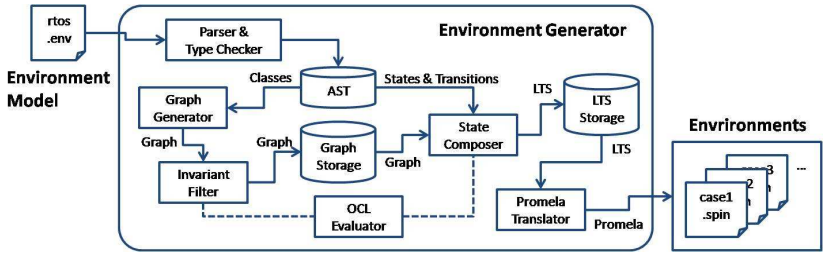

Fig. 8. Architecture of the environment generator

The first attempt to generate the environments failed because the computation time took too long. This inefficiency was attributed to the exponential growth in the number of association variations. In order to make the environment generation efficient, we removed some of the associations by heuristics. Specifically, from the three associations, i.e., `Task` to `Task`, `Task` to `Resource`, and `Resource` to `Task`, we removed the first and the third association by the following approach.

Firstly, we removed the association from `Task` to `Task` by over-approximating the environment. Originally, we included this association to define the variation of function call relationships between tasks. For example, if a task T1 has a link to a task T2, T1 can call `ActivateTask()` to T2, but cannot if it does not have the link. We checked this condition in the guard condition of `ActivateTask()`. By removing this association and the condition, we made it possible for any tasks to call the function to any tasks. This is an over-approximation of the environment, i.e., the execution paths of this modified environment retains the execution paths of the original environment. If we can prove that the assertions are satisfied in the over-approximated model, we can conclude that the assertions are also satisfied in the original model. This is because the assertions are universal properties which must be satisfied in every state of the model. Of course, the over-approximation can cause false positive errors.

Secondly, we removed the association from `Resource` to `Task` making use of an invariant. This association defines that the multiplicity from `Resource` to `Task` is at least one. As we have the association from `Task` to `Resource`, we can express the same constraint based on this association by the invariant on `Resource`: `Task->exists(t|t.res->contains(self))`. It means that for a resource (`self`), there exists at least one task which has a link to the resource. In fact, the number of environments does not change by this modification.

By removing the two associations, we were able to generate all the environments up to 6 objects within 10 minutes. Still, we cannot generate the environments for the cases of more than 6 objects in a reasonable time. To make the generation more scalable, we need to device an algorithm to compute the set of object graphs, which satisfy invariants, directly from the class model. To realize this, we are currently considering the use of SAT and SMT solvers [14, 4].

R/T	1	2	3	4
0	4 (0.0s)	6 (0.1s)	4 (0.1s)	1 (0.1s)
1	10 (0.1s)	40 (0.3s)	55 (1.7s)	26 (4.5s)
2	10 (0.1s)	95 (1.6s)	245 (20.1s)	196 (153.9s)
3	5 (0.2s)	100 (7.1s)	425 (203.8s)	N/A
4	1 (1.0s)	39 (67.3s)	N/A	N/A

Table 1. The number of generated environments (CPU:2.4GHz, Memory:4.0GB)

5.2 Model checking

We conducted model checking on some of the generated environments. We selected a representative environment from each case in Table. 1. For example, for the 95 cases in T=2 and R=2, we selected the case No. 95/2=47. Thanks to the structural decomposition of the environment, we were able to check all of them without causing state explosion. The results of the model checking are listed in Table. 2 with the structures of object graphs.

The table shows that 6 among 17 cases were unsuccessful. All the failures were caused by assertion violations. Specifically, the runtime priority of a task was inconsistent between the environment and the RTOS. In our method, we can make use of the table to conduct difference analysis to find the source of a bug. By overlooking the table, we can see that the check fails only if there are at least two tasks. With a closer look, we can notice that the check fails only if there is at least one task which is linked with more than one resources. Based on this information, we can presume that the resource handling of the RTOS is incorrect, especially when a task tries to acquire more than one resources.

By examining the counter example traces output by SPIN, we found that the bug was contained in the function `GetResource()`. It is a function called from a task when the task acquires a resource. Correctly, the function must raise the runtime priority of the task to the resource priority only when the runtime priority is lower than the resource priority. (This mechanism is called ceiling priority protocol.) But the function changed the runtime priority everytime the task acquires a resource regardless of the resource priority. This bug caused the inconsistency of the runtime priority between the environment and the RTOS. (For example, if T1 first acquires R1 of priority 4 and then acquires R2 of priority 3, the environment expect the runtime priority of T1 to be 4. In the RTOS, however, it was incorrectly changed to 3.) This inconsistency occurs only when a task tries to acquire more than one resources. This result coincides with our presumption based on the difference analysis.

When model checking in SPIN, it is not always easy to pinpoint the source of the bug only from the information of the counter example traces. Our method, however, can provide additional information as a form of a table by which we can conduct difference analysis based on graph structures. This is one of the advantages of structurally decomposing the whole environment into individuals.

T,R	No.	Task.pr				Task.res				Resource.pr				Result
		T1	T2	T3	T4	T1	T2	T3	T4	R1	R2	R3	R4	
1,0	2	2				[]								◯
1,1	5	2				[1]				3				◯
1,2	5	1				[1,2]				1	4			×
1,3	2	1				[1,2,3]				1	2	4		×
1,4	1	1				[1,2,3,4]				1	2	3	4	×
2,0	3	2	3			[]	[]							◯
2,1	20	3	4			[1]	[]			4				◯
2,2	47	2	3			[1]	[2]			2	4			◯
2,3	50	1	3			[1,2,3]	[3]			1	3	4		×
2,4	19	1	2			[1,2,3,4]	[2,3]			1	2	3	4	×
3,0	2	1	2	4		[]	[]							◯
3,1	27	1	2	4		[1]	[1]	[]		2				◯
3,2	122	1	2	3		[1]	[]	[2]		1	3			◯
3,3	212	1	2	3		[2]	[1]	[3]		2	3	4		◯
4,0	1	1	2	3	4	[]	[]	[]	[]					◯
4,1	13	1	2	3	4	[1]	[]	[1]	[]	3				◯
4,2	98	1	2	3	4	[1]	[1]	[1,2]	[]	3	4			×

Table 2. Results of model checking

6 Discussion

6.1 Coverage of environment variations

In the experiment, we set the limit of the number of tasks and resources (and also the range of priorities) to 4. This is, however, only a part of possible variations defined in the OSEK/VDX RTOS specification. As it is impossible to cover all the cases due to the computation time, we need criteria as to how far we should extend the variation of the environment model. In our experience, crucial errors concerning the behavioral logic of the system can be discovered even with a small number of objects, and the errors newly discovered by increasing the number of objects are only those concerning system boundaries (such as generating objects which exceeds the limit of an array). So, one of the criteria should be to clarify the important properties which must be satisfied by the system and cover at least the variations which can capture the satisfaction of the properties. For example, the important properties about the behavioral logic of RTOS are: (1) Without resources, the task of higher priority must be executed before that of lower priority, (2) If a task occupies multiple resources, its runtime priority is set to the maximum priority of the resources, (3) With resources, the task of higher runtime priority must be executed before that of lower runtime priority. To check these properties, we need at least 2 tasks and 2 resources, i.e., (1) requires 2 tasks, (2) requires 1 tasks and 2 resources, and (3) requires 2 tasks and 2 resources. Like this, we need to identify the necessary minimum variations

depending on the properties to check and define the parameter ranges sufficiently wide to cover the variations.

Furthermore, lots of errors can be discovered even by checking some of the environments instead of checking all of the environments. In fact, we were able to discover a crucial error of the RTOS logic by checking only 17 environments among a total of more than 1200 environments. So, we consider it also effective to generate a limited number of environments from all the variations by sampling at regular intervals or completely at random. This is especially effective in the early stage of development where many easy errors are contained.

As for behavioral coverage, we further need to conduct verification for abnormal execution sequences and interrupt handling as well as the normal execution sequences which we verified in the experiment. To verify interrupt handling, we need to extend the environment model so that it can deal with multiple processes since interleavings occur between the execution of tasks and the interrupt handler.

6.2 Parallelization of model checking

In order to check the large amount of generated environments efficiently, we are considering the parallelization of all the checks by distributing them on a PC cluster. Since all the environments are structurally different from each other, they can be checked independently of others. To realize this, we need to address two problems. The first one is load-balancing, i.e., how to distribute all the environments equally to each PC. For this problem, we consider it effective to distribute them based on the length of Promela files. This is because most of the time for checking an environment is occupied by the compilation of the Promela file of the environment. The second one is data-mining, i.e., how to retrieve useful information from the large amounts of check results and display it to users. For the retrieval of information, we consider it effective to store the results in a relational database. This makes it possible for us to retrieve necessary information by issuing query based on environment structures. For the display of the results, we consider it effective to make use of a table as we did in the experiment. The table gives us a birds-eye-view on all the results. This makes it visually easy for us to conduct difference analysis on the results. We further need to device a more effective way to display the table.

As well as model checking, we can also parallelize the generation of environments. As we are generating environments based on the graph counter, we can parallelize it by dividing the counter and conducting enumeration in each PC.

7 Related work

Many work propose methods to verify UML models in SPIN by translating statechart models into Promela [17, 7, 12, 6] Our method also applies model checking to statechart models, but our motivation is totally different from these works in

that we are using statechart models for describing environments, not the target system itself.

O. Tkachuk, et al. [18] proposes Bandera Environment Generator (BEG) which automatically generates the environment for the verification of Java programs in Bandera. It has been applied to commercial software [19] and has also used as the core tool for the environment generator for web application domain [15]. In BEG, the environment is generated from the specifications of the environment written by the user, called environment assumptions, or by analyzing the programs which implements the environment. The environment assumptions are described as the sequences of the method calls in the form of regular expressions. This approach corresponds to describing a single instance of the environment model in our method. In our method, we can describe a set of the instances as a class model and automatically generate all the possible instances based on the variations in the model.

P. Parizek, et al. [10] proposes a method to verify Java components by Java PathFinder (JPF) and a protocol checker. The protocols of the components are defined by ADL (Architecture Description Language) from which the environment for the components is constructed. This method allows to describe the environment structure by ADL. Compared to this work, our method further describe t he variation of the structure using the parameters in the class model.

J. Penix, et al. [13] [11] verifies the time partitioning of DEOS RTOS by SPIN. M. Dwyer et al. [5] verifies parital systems described in Ada by translating them into SPIN. In these works, environments are obtained by filtering a universal environment with assumptions described in LTL. This approach is effective when the assumptions can be described simply, but shows weakeness when describing precise behavior of environments due to the accumulation of complex LTL assumptions. Compared to these works, we describe the specific behavior of an environment from the begining using statechart models instead of incrementally refining the universal environment by assumptions. The use of statechart models facilitates the desctiption of environements because the abstraction level is lifted to the familiar level for users.

Concerning distributed parallel model checking, J. Barnat et al. [3] developed a tool for parallel LTL model checking and reachability analysis for multi-core systems and clusters. As we explained in discussion, our method has the potential to be parallelized. Compared to this work which parallelizes search algorithms, our parallelization is based on data, i.e., we statically divide the whole environment by changing its data settings.

8 Conclusion

In this paper, we presented a UML-based method for modeling and generating environments for model checking embedded systems. We presented a tool to automatically generate Promela/SPIN scripts from the environment model. As an application to a practical system, we conducted a verification of an OSEK/VDX RTOS design model. In the experiment, we were able to generate sufficient varia-

tions of environments efficiently for checking crucial properties of RTOS. We also identified the effectiveness of difference analysis based on the environment structures. Future work is to develop a distributed parallel model checking framework based on our method.

References

1. OSEK/VDX. http://portal.osek-vdx.org/.
2. Toshiaki Aoki. Model Checking Multi-Task Software on Real-Time Operating Systems. In *ISORC*, pages 551–555. IEEE Computer Society, 2008.
3. Jiří Barnat, Luboš Brim, and Petr Ročkai. DiVinE 2.0: High-Performance Model Checking. In *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, pages 31–32. IEEE Computer Society Press, 2009.
4. Leonardo Mendonça de Moura, Bruno Dutertre, and Natarajan Shankar. A Tutorial on Satisfiability Modulo Theories. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2007.
5. Matthew B. Dwyer and Corina S. Pasareanu. Filter-Based Model Checking of Partial Systems. In *SIGSOFT FSE*, pages 189–202, 1998.
6. Diego Latella, István Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.
7. Johan Lilius, Ivan Porres, Ivan Porres Paltor, Turku Centre, and Computer Science. vUML: a Tool for Verifying UML Models. pages 255–258, 1999.
8. Jeff Magee and Jeff Kramer. *Concurrency: State models & Java programs.* Wiley, 1999.
9. OMG. Unified Modeling Language. http://www.omg.org/.
10. Pavel Parizek and Frantisek Plasil. Partial Verification of Software Components: Heuristics for Environment Construction. In *EUROMICRO-SEAA*, pages 75–82. IEEE Computer Society, 2007.
11. Corina S. Pasareanu. DEOS Kernel: Environment Modeling using LTL Assumptions. Nasa ames technical report nasa-arc-ic-2000-196, 2000.
12. Patrizio Pelliccione, Paola Inverardi, and Henry Muccini. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *IEEE Trans. Software Eng.*, 35(3):325–346, 2009.
13. 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.
14. Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.
15. 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.
16. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.
17. Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electr. Notes Theor. Comput. Sci.*, 55(3), 2001.
18. 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.

19. Oksana Tkachuk and Sreeranga P. Rajan. Application of automated environment generation to commercial software. In Lori L. Pollock and Mauro Pezzè, editors, *ISSTA*, pages 203–214. ACM, 2006.
20. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

A Formal definitions and algorithms

A.1 Environment model

Definition 1 (Environment model). *An environment model EM is defined as:*

$$EM = (I, \mathcal{C}, \mathcal{S})$$

I is a target system. $\mathcal{C} = \{C_1, C_2, ...\}$ is the set of classes. $\mathcal{S} = \{SC_1, SC_2, ...\}$ is the set of statechart models.

Definition 2 (Target system). *The target system I is defined as:*

$$I = (Fun, Ref, Arg)$$

The sets Fun and Ref are the set of trigger functions and reference functions, respectively. The mapping $Arg : F \times 2^{Val}$ relates a function to the domain (variation) of its argument. (For simplicity, we present the definition where a function has only one argument.) The set Val is the set of values.

Definition 3 (Classes). *The class $C_i \in \mathcal{C}$ is defined as:*

$$C_i = (X_i, Y_i, V_i, Size_i, Dom_i, Def_i, Inv_i, Assr_i)$$

The sets X_i and Y_i are the sets of attributes and associations, respectively. We define the union $X_i \cup Y_i$ as $U_i = \{u_{i1}, u_{i2}, ...\}$, and call them members of the class C_i. The set $V_i = \{v_{i1}, v_{i2}, ...\}$ is the set of variables. $Size_i \in N$ is the number of objects which instantiate from the class. The mapping $Dom_i : U_i \rightarrow 2^{Val}$ relates a member to the set of values which represents the domain (variation) of the member. We represent each element in $Dom_i(u_{ij})$ as $d_{ijk}(k = 1, ...)$. The mapping $Def_i : V_i \rightarrow Val$ relates a variable to its default value. We represent each element in $Def_i(v_{ij})$ as e_{ij}. The expression $Inv_i, Assr_i \in Exp$ are an invariant and an assertion. Exp is the set of expressions ($Val \subset Exp$).

In this definition, we omit the definition of multiplicities of associations. The variation of links for an association is directly defined by Dom_i.

Definition 4 (Statechart models). *Let S be the set of states. For the class C_i, the statechart model SC_i is defined as:*

$$SC_i = (S_i, s_{0i}, A_i, T_i)$$

The set $S_i \subset S$ is the set of states. The state $s_{0i} \in S_i$ is the initial state. The set A_i is a set of actions. The set T_i is the set of transitions. For a transition

$(s_1, s_2, g, f, a, st) \in T_i$, $s_1, s_2 \in S_i$ are the source and destination states, respectively. $g \in Exp$ is a guard condition. $f \in Fun$ is a trigger function. $a \in A_i$ is an action. st is the set of synchronous transitions. For a synchronous transition $(t_1, t_2, x) \in st$, $t_1, t_2 \in S$ are the source and destination states, respectively. $x \in Exp$ is the expression representing the synchronized objects.

A.2 Generation of object graphs

Definition 5 (Object graphs). *Let $O_i = \{o_{i1}, o_{i2}, ...\}$ be the set of objects which instantiate from the class c_i ($|O_i| = Size(c_i)$). Let $\mathcal{G}$ be the set of object graphs. An object graph $G \in \mathcal{G}$ is defined as:*

$$G = \{g_{ijk} | i = 1..|C|, j = 1..|O_i|, k = 1..|U_{ij}|\}$$

The value g_{ijk} represents the value of the member u_{ik} of the object o_{ij}.

The set of object graphs $\mathcal{G}$ is computed as follows. Firstly, we define the graph counter $Z = \langle z_1, z_2, ... \rangle$ as a vector of length $M = \sum_{i=1}^{|C|} (|O_i| \times |U_i|)$. The correspondence between object members and graph counter elements are defined by the function $Pos(i, j, k) = (\sum_{n=1}^{i-1} |O_n| \times |U_n|) + |U_i| \times (j - 1) + k$. If $Pos(i, j, k) = p$, z_p corresponds to the member u_{ijk}. For each z_i, we define its maximum value as Max_i. If $Pos(i, j, k) = p$, Max_p is equal to $|Dom(u_{ik})|$.

Then, we define a function *GetGraph* which generates an object graph from the graph counter.

$GetGraph(Z) = (G, Z')$

where

$$G_{ijk} = d_{ijm} \quad (m = L[Pos(i, j, k)], i = 1..|C|, j = 1..|O_i|, k = 1..|U_i|)$$

$$Z' = \begin{cases} \langle z_1, ..., z_M + 1 \rangle & \text{if } z_M < Max_M \\ \langle z_1, ..., z_i + 1, 1, ..., 1 \rangle & \text{if } z_i < Max_i,\ z_j = Max_j\ (j = i + 1..M) \\ \langle 1, ..., 1 \rangle & \text{if } z_i = Max_i\ (i = 1..M) \end{cases}$$

G and Z' are the generated object graph and the next graph counter, respectively. The last case of Z' means that the counter returns to $\langle 1, ..., 1 \rangle$ when it reaches maximum. By this, we know the end of counting.

Finally, the set $\mathcal{G}$ is obtained the following algorithm. In the algorithm, the mapping $Eval_G[o] : Exp \rightarrow Val$ relates an expression to a value which is obtained by evaluating the expression in the context of a graph G and an object o.

1. Let $\mathcal{G} = \{\}$ and $Z = \langle 1, ..., 1 \rangle$.
2. Let $(G, Z) = GetGraph(Z)$
3. If G satisfies invariants, i.e., $Eval_G[o_{ij}](Inv_i)$ is a value representing true for all i and j, let $\mathcal{G} = \{G\} \cup \mathcal{G}$.
4. If $Z \neq \langle 1, ..., 1 \rangle$, goto 2.

A.3 Composition of statecharts

Definition 6 (Labelled transition systems). *An LTS L is defined as:*

$$L = (P, p_0, Q, R, B)$$

The set $P = \{p_0, p_1 ...\}$ is a set of composite states. In a composite state p, the state of the object o_{ij} and the value of the variable v_{ijk} are defined as $p[i, j, 0]$ and $p[i, j, k]$, respectively. (Note that $k = 1...$) The state p_0 is the initial state. The set Q is the set of labels. For a label $(f, w) \in Q$, f and w are a trigger function and its argument, respectively. The set R is the set of transitions. For a transition $r = (t_1, t_2, q) \in R$, t_1, t_2, and q are a source state, a destination states, and a label, respectively. The set $B = \{b_0, b_1, ...\}$ is the set of assertions. Each b_i is the assertion defined for the state p_i.

For an object graph G, an LTS L is computed by the following algorithm. In the algorithm, the mapping $Eval_G[o][p][w] : Exp \rightarrow Val$ relates an expression to a value which is obtained by evaluating the expression in the context of a graph G, a state p, an object o, and an argument w. The mapping $AEval_G[o][p] : Exp \rightarrow Exp$ relates an assertion to an expression which is obtained by evaluating the assertion in the context of a graph G, a state p, and an object o. (The result expression may contain reference functions.) The mapping $Exec_G[o][p][w] : A \times V \rightarrow Val$ relates an action and a variable to the value of the variable which is obtained by evaluating the action in the context of a graph G, a state p, and an argument w. For simplicity, we present the algorithm for the case where the expression of a synchronous transition always evaluates to a single object.

1. Let $P = Q = B = \{\}$.
2. Define the initial state p_0 such that $p_0[i, j, 0] = s_{0i}$ and $p_0[i, j, k] = Def_i(y_{ik})$.
3. Let $p = p_0$ and $P_i = \{p_0\}$ (p is a temporal variable).
4. For each object o_{ij}, for each transition $(p_{ij0}, s, g, f, a, st) \in T_i$, and for each argument $w \in Arg(f)$, do the following steps.
   (a) If the guard condition g is true, i.e., the expression $Eval_G[o_{ij}][p][w](g)$ is a value representing true, create a new state q with $q[m, n, 0]$ defined as follows:
      – If $m = i$ and $n = j$, then s.
      – If there exists a synchronous transition $(x, t_1, t_2) \in st$ such that the target object x is o_{mn}, i.e., the expression $Eval_G[o_{ij}][p][w](x)$ is a value representing o_{mn} and $p[m, n, 0] = t_1$, then t_2.
      – Otherwise, $p[m, n, 0]$.

      For all cases, $q[m, n, l]$ is defined as the value $Exec_G(o_{mn}, a, v_{ml})$.
   (b) Let $Q = \{(f, w)\} \cup Q$ and $R = \{(p, q, (f, w))\} \cup R$.
   (c) If $q \notin P$, let $P = \{q\} \cup P$ and $p = q$, and go to 4.
5. For each $p_i \in P$, let $b_i = \bigwedge_{j,k} AEval_G[o_{jk}][p_i](Assr_j)$ and $B = \{b_i\} \cup B$.