JAIST Repository
https://dspace.jaist.ac.jp/

Title	Automatic Generation of Model Checking Scripts based on Environment Modeling
Author(s)	Yatake, Kenro; Nishibata, Hirokazu; Aoki, Toshiaki
Citation	Research report (School of Information Science, Japan Advanced Institute of Science and Technology), IS-RR-2010-001: 1-8
Issue Date	2010-02-10
Type	Journal Article
Text version	publisher
URL	http://hdl.handle.net/10119/8839
Rights	
Description	リサーチレポート（北陸先端科学技術大学院大学情報科学研究科）

JAIST
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Automatic Generation of Model Checking Scripts based on Environment Modeling

Kenro Yatake, Hirokazu Nishibata, Toshiaki Aoki

2010/2/10

IS-RR-2010-001

Automatic Generation of Model Checking Scripts based on Environment Modeling

Kenro Yatake, Hirokazu Nishibata, and Toshiaki Aoki
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa, Japan, 923-1292
{*k-yatake, s0710054, 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 surrounding the system. In this paper, we present a UML-based method to model the environment and to generate environment instances from the model. In our method, we can flexibly model the variation of the environment structures and the sequences of the function calls using class diagrams and statechart diagrams. We also present a tool to automatically generate Promela/Spin scripts from the environment model. In this paper, we explain the details of our method and the verification of an RTOS design model using the tool.**

I. Introduction

Recently, model checking [3][4] is drawing attention as a technique to improve the reliability of the software systems. Especially, they are widely applied to the verification of embedded systems. The major characteristics of embedded systems is their reactiveness, i.e., they operate by the stimulus from the outside world. For example, network printers operate by printing requests from the client hosts. For another 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 outside world. This is called an *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, the property description tends to become complicated because it must characterize all the sequences with a single predicate. Furthermore, it often suffers state explosion because all the sequences are checked at a time. Another approach is to call specific sequences of the 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 likely to be able to avoid state explosion.

We consider the latter approach is more realistic. But 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 a variety of values for their priorities. The number of their variation becomes so huge that we cannot construct all of them by hand. Therefore, we need a method to model the structural variation of the environment and to automatically generate each environment from the model. To our knowledge, there are no established methods to support this on the design level.

To satisfy the need, we propose a method to model the environment based on UML (Unified Modeling Language) [1]. In our method, we can flexibly model the variation of the environment and the sequences of the function calls using class diagrams and statechart diagrams. We also implemented a tool to automatically generate Promela/Spin scripts from the model. As an experiment, we applied the tool to the verification of an OSEK/VDX RTOS [5] design model. 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 generation of environments from the model. In section 5, we explain the implemented tool. In section 6, we explain the verification experiment. In section 7, we discuss the effectiveness of our method. In section 8, we give a conclusion and future work.

II. Approach

The system like an RTOS does not operate by itself, but operates by getting its functions called from outside. So, in order to verify its behavior, we need to prepare an environment to call the functions, and check if the system operates correctly for each of the function calls. Fig.1 summarizes this idea. It shows an example of an RTOS. It implements data structures such as a task list `tsk[]` and a ready queue `ready[]` and provides API functions such as `ActivateTask()` and `TerminateTask()`. If these functions are called, it manages the scheduling of tasks based on their priorities. To verify its behavior, we prepare an environment consisting of two tasks T1 and T2 (T2's priority is higher than T1's). It describes a sequence of function

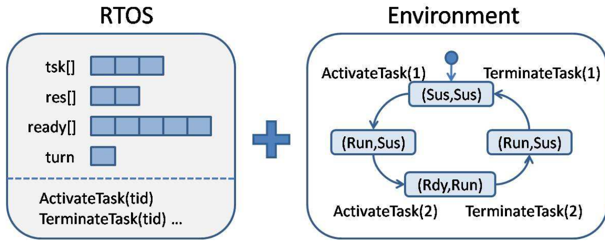

Figure 1. Model checking with an environment

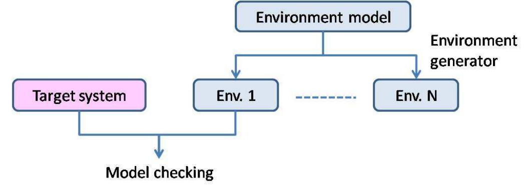

Figure 2. Environment modeling method

calls to the RTOS and state transitions of the tasks expected by the calls. For example, if the function ActivateTask() is called to T1, it is expected to become running. Then, if the same function is called to T2, T1 and T2 are expected to become ready and running, respectively (T2 preempts T1's execution). By applying model checking to the RTOS in combination with the environment, we can verify that the RTOS satisfies this expectation. 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 value of T1, and the value of the variable `turn` (representing the running task) must be equal to the identifier value of T2. In this way, we can verify the target system using an environment.

The problem is that this environment is only one of the cases of the huge number of environment variations. We need to verify it for all the variations with respect to the number of tasks and resources, the patterns of the priority values, the patterns of reference relationships, and so on. 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 it is likely to end up in state explosion.

To cope with this problem, we introduce a model to describe the environment variations and automatically generate all the environment instances from the model. Fig. 2 summarizes this idea. To verify the target system, we first construct an *environment model*. Then, we input the model to the *environment generator* and obtain environment instances. Finally, we apply model checking to the target system in combination with each generated instances. The advantage of this approach is that we can avoid state explosion by dividing the whole environment into individual environment which can be checked in a relatively small state space.

The environment model is based on UML. In a class diagram, we describe the structural variation of the environment with respect to the elements such as the number of objects, attribute values and association multiplicities. In statechart diagrams, we describe the sequences of function calls and the state transitions expected for them. As a model checker, we use Spin. The environment generator inputs an environment model as a text file and outputs the instances as Promela scripts.

III. Environment models

In this section, we introduce the definition of the environment model. It is based on UML, but it also contains our original notations. We begin with an overview with the example of RTOS, and then present the formal definition. The RTOS is based on the OSEK/VDX specification.

A. Overview

1) Class diagram: Fig.3 shows the class diagram of the environment for RTOS. The class diagram consists of a class representing the target system and classes representing the environment. In the figure, the class `RTOS` is the target system and the classes `Task` and `Resource` are the environment classes.

The target class defines two kinds of functions as the interface with the target system. The functions labeled with `action` are the functions to drive the target system (called *driver functions*). For example, `ActivateTask(tid)` is the action to activate the task of ID `tid`. In order to define the variation of the function call, the argument of the action is defined with the range like `tid:TD` (`TD` is defined as `{1,2}`). In this case, two variations of the function call are considered : `ActivateTask(1)` and `ActivateTask(2)`. The functions labeled with `info` are the functions to refer to the internal values of the target system (called *reference functions*). These functions are used to define assertions, which is explained later in this section.

The environment classes are defined with attributes and associations. Attributes are also defined with the ranges like `pr:{1,3}` (representing the priority). In this case, two variations of the attribute are considered for a `Task` object. Associations are defined with multiplicities like `(0,TN)` (`TN` is defined as `2`). It is a pair of the minimum and the maximum number of objects linked with an object. Links are generated between objects so that they cover all the patterns in this range. The multiplicities of the associations from the target class to the environment classes represent the number of objects which instantiate from the environment classes. In this case, two `Task` objects and one `Resource` object are created.

Invariants and assertions are defined for the environment classes. They are written in OCL (Object Constraint Language) [2]. Invariants define the constraint on the structure

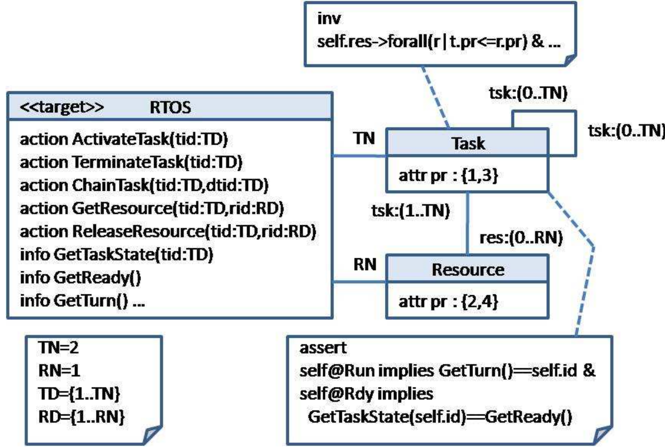

Figure 3. The class diagram for the RTOS environment

of objects. The invariant for the class `Task` means: "All the `Resource` objects linked to a `Task` object have the priorities not lower than the `Task` object." Assertions define the predicate to check in each state of objects. To refer to the values in the Promela script of the RTOS, the `info` functions are used. The assertion for the class `Task` means: "If a `Task` object is in the state `Run`, the value of the function `GetTurn()` must be equal to the identifier of the object (`id` is a built-in attribute), and if it is in the state `Rdy`, the value of the function `GetTaskState()`, whose argument is the identifier of the object, must be equal to the value of the function `GetReady()`". The `info` functions refer to the variable `turn`, the struct member `tsk[self.id].stat` (representing the task state) and the constant `READY` (representing the state value for ready), respectively.

2) Statechart diagrams: In statechart diagrams, we define the state transitions of the environment objects expected for the function calls of the target system. As we stated in introduction, we only describe specific sequences of function calls of the target system. Fig.4 shows the statechart diagram of the class `Task`. It describes the normal execution sequences of RTOS. A transition is triggered by a function call. For example, the transition (1) means: "When the action `ActivateTask` is called for the `Task` object in the state `Sus` (Suspended) and if there are no other tasks in the state `Run`, the object transits to the state `Run` (Running)." The expression in `[ ]` is the guard condition written in OCL. In the model, typical expressions are defined as functions like `ExRun()=Task->exists(t|t@Run)` (checks if there exists a `Task` object in the state `Run`). The OCL expressions in our model also contain our original syntax, but it is basically a subset of OCL containing the set operations and the state reference.

A set of *synchronous transitions* can be attached to a transition. A synchronous transition defines the transitions of other objects which occur synchronously with the transition of the self object. For example, the transition (2) defines the synchronous transition `Rdy->Run {GetRdyMax()}`. This means: "Along with the transition of the self object, the `Task` object obtained by the function `GetRdyMax()` (the task which is in the state `Rdy` (Ready) and has the maximum priority) transits from the state `Rdy` to `Run`. The OCL expression in { } represents the synchronized objects of the transition. We adopted this syntax instead of the usual event passing simply because it is more direct for describing synchronization among multiple objects. We do not consider asynchronous transitions.

Let us note here about the semantic detail of the synchronous transition. If the OCL expression of a synchronous transition is evaluated to a set of objects, one of them is chosen non-deterministically as the synchronized object. For example, if `GetRdyMax()` returns a set of two tasks T1 and T2, two transitions are generated. One is to transit T1 from `Rdy` to `Run` and the other is to transit T2 from `Rdy` to `Run`. This non-determinism approximates the ideal behavior of the environment. See 7.1 for the details.

B. Formal definition

1) Class diagrams: The class diagram is defined as follows:

$$
\begin{aligned}
CD = (&C, At, As, Attr, Assoc, Size, Dom, Mult,\\
&Inv, Assr),\\
&Attr : C \to 2^{At},\ Assoc : C \to 2^{As},\ Size : C \to N,\\
&Dom : At \to 2^{Int},\ Mult : As \to C \times N^2,\\
&Inv : C \to Exp,\ Assr : C \to Exp
\end{aligned}
$$

The set C is the set of the environment classes. The sets At and $Assoc$ are the sets of attributes and associations, respectively. The mappings $Attr$ and $Assoc$ relate a class to its attributes and associations, respectively. The mapping $Size$ relates a class to the number of objects from the class. The mapping Dom relates an attribute to its domain, or the set of values. We consider only integers for the values of attributes (for the compatibility with Promela). The mapping $Mult$ relates an association to the destination class and the multiplicity. For an association as with $Mult(as) = (c, min, max)$, the class c is the destination class and the natural numbers min and max are the minimum and maximum number of the multiplicity. The mapping Inv relates a class to the invariant expression in OCL. The mapping $Assr$ relates a class to the assertion expression in OCL. The set Exp is the set of OCL expressions. We abstract away from their concrete syntax.

The target system is defined as a set of functions as follows:

$$
\begin{aligned}
&TS \equiv (A, I, ArgD, ArgN)\\
&F \equiv A \cup I,\ ArgN : F \to N,\ ArgD : F \times N \to 2^{Int}
\end{aligned}
$$

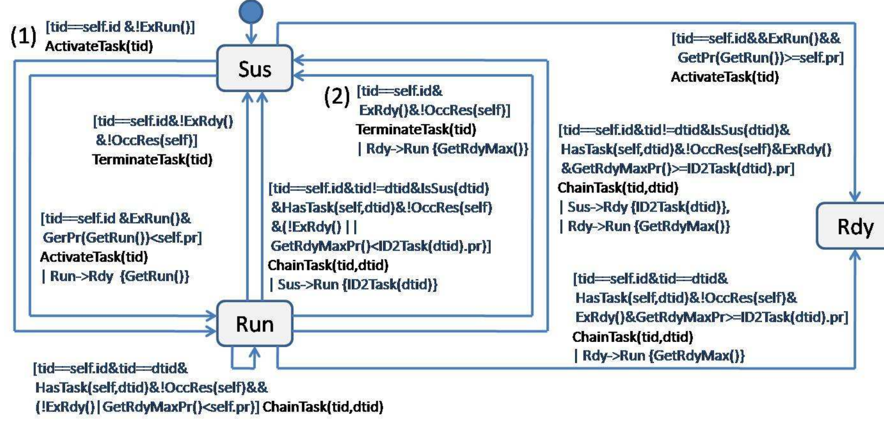

Figure 4. The statechart diagram of the class Task

The set A is the set of driver functions. The set I is the set of reference functions. The set F is defined as a union of these functions. The mapping $ArgN$ relates a function to the number of its arguments. The mapping $ArgD$ relates a function and a natural number n to the domain of its n-th argument.

2) Statechart diagrams: Let S be the set of states. For the class $c_i \in C$, the statechart diagram SD_i is defined as follows:

$$
\begin{aligned}
&SD_i \equiv (S_i, init_i, T_i) \\
&S_i \subseteq S \ (S_i \cap S_j = \phi,\ i \neq j),\ init_i \in S_i, \\
&T_i : S_i \times S_i \times Exp \times A \times 2^{S \times S \times Exp}
\end{aligned}
$$

The set S_i is the state sets of the class c_i and the state $init_i$ is the initial state. The set T_i is the set of transitions. For the transition $(s_1, s_2, c, a, st) \in T_i$, the states s_1 and s_2 are the source and destination states, respectively. The predicate c written in OCL is the guard condition. The action a is the action to trigger this transition. The set st is the set of synchronous transitions. For the synchronous transition $(t_1, t_2, x) \in st$, the states s_1 and s_2 are the source and destination states, respectively. The OCL expression x defines the set of the synchronized objects. Actually, states can take arguments such as `Occ(tid)` (the state of a `Resource` object occupied by the `Task` object of `tid`), but we omit its formalization for simplicity.

IV. Generation of environments

A. Overview

Generation of environment instances is done in three steps: (1) Generation of object graphs, (2) Composition of statechart diagrams, and (3) Translation into Promela. The explanation of (3) is left to the next section.

1) Generation of object graphs: The object graph is the graph structure with nodes represented by objects and edges represented by the links which instantiate from associations. Each node has attribute values as its data. The set of object graphs is generated so that it covers all the variations of attribute values and association multiplicities. Logically, this is done in the following three steps:

Firstly, we compute the product of the attribute variations. In the example, the attribute `pr` of the class `Task`, and also of `Resource`, has two variations. As there are two objects from the `Task` class and one object from the `Resource` class, the number of the attribute variations as a whole become $2^2 \times 2^1 = 8$. Then, we compute the product of the association variations. Let T1 and T2 be the two `Task` objects and R1 be the single `Resource` object. Under the multiplicities of the associations, both T1 and T2 can link to any of the three objects T1, T2 and R1 ($2^3 = 8$ patterns), and R1 links to at least one of the two objects T1 and T2 (3 patterns). The number of the graph structure satisfying this constraint is $8^2 \times 3 = 192$. Finally, from the total of $8 \times 192 = 1536$ graphs, we retain only the graphs which satisfy invariants. This results in 104 graphs.

In the actual implementation, we compute the graphs as a *stream*, or generate each graph one after another by moving the parameters in the class diagram within the range of their variation. This can reduce the computation space to the order of $(|At| + |As|) \times |O|$.

2) Composition of statechart diagrams: For each object graph, we compose the statechart diagrams of all the objects in the graph. The result of the composition is an LTS (Labelled Transition System). Fig.5 shows an example of an object graph and its LTS.

In the LTS, each state is represented by the tuple of the states of all the objects like `(Sus,Sus,Fre)`. Transitions in statechart diagrams are added to LTS if their guard conditions are evaluated to true in the LTS state and the object graph. For example, the transition from `(Sus,Sus,Fre)` to `(Run,Sus,Fre)` by the action `ActivateTask(1)` (`AT(1)`) is added because the guard condition of the transition (1) in Fig 4 becomes true for

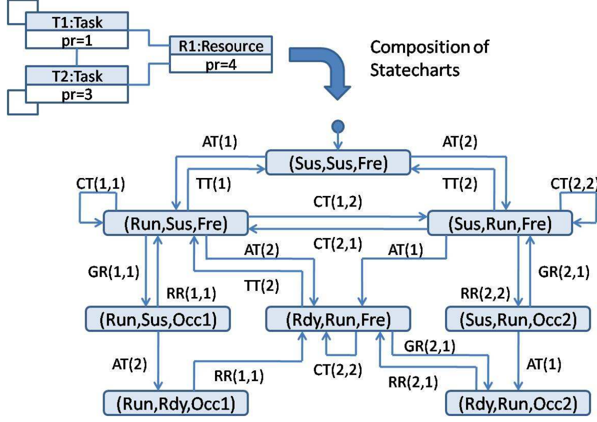

Figure 5. The object graph and its LTS

the object T1 (`id==1`) and the action argument `tid==1` in the state `(Sus,Sus,Fre)`. If a transition has synchronous transitions, the target objects are obtained by evaluating the OCL expression and they transit along with the transition of the self object. For example, in the transition from `(Rdy,Run,Fre)` to `(Run,Sus,Fre)` by the action `TerminateTask(2)` (`TT(2)`), the target object T1 transits from `Rdy` to `Run` along with the self object T2 transiting from `Run` to `Sus`.

It could be possible to translate all the statechart diagrams directly into Promela and leave the composition to SPIN. But we do not take this approach because the transitions attached with OCL expressions in our model are difficult to express directly in Promela. So, we compose the statechart diagrams at this point outside SPIN.

B. Formal definition

In the following, we represent the elements in the sets $Attr(c_i)$ and $Assoc(c_i)$ as at_{ij} ($j = 1...|Attr(c_i)|$) and as_{ij} ($j = 1...|Assoc(c_i)|$).

1) Generation of object graphs: Let OG be the set of the object graphs generated from the class diagram. We define each element $G_i \in OG$ as follows:

$$G_i = (O, Val_i, Link_i)$$
$$Val_i : At \to O \to Int,\ \ Link_i : As \to O \to 2^O$$

The set O is the set of objects. This set is common to all the object graphs. We represent each element in O as o_{ij} ($i = 1...|C|$, $j = 1...Size(c_i)$). The object o_{ij} is the j-th object in the class c_i. The mapping Val_i relates an attribute and an object to its value. The mapping $Link_i$ relates an association and an object to the set of the destination objects. The elements of the mappings Val_i and $Link_i$ are defined as the following computation.

Firstly, we compute the product of attributes and associations:

$$X \equiv \prod_{i=1}^{|C|} \prod_{j=1}^{|Attr(c_i)|} Dom(at_{ij})^{|c_i|},$$
$$Y \equiv \prod_{i=1}^{|C|} \prod_{j=1}^{|Assoc(c_i)|} L_{ij}^{|c_i|}$$

where

$$L_{ij} \equiv \{s | s \subset O_k \wedge min \leq |s| \leq max\},$$
$$O_k \equiv \{o_{kj} | 1 \leq j \leq |c_k|\},$$
$$(c_k, min, max) \equiv Mult(as_{ij})$$

Then, we compute the product of the both:

$$Z \equiv \{Z_1, ..., Z_{|X||Y|}\} \equiv X \times Y$$

Finally, we define the elements in Val_i and $Link_i$ in terms of Z as follows:

$$Val_i(at_{jk}) \equiv \bigcup_{m=1}^{|c_j|} \{o_{jm} \mapsto [[[[Z_i]_1]_j]_k]_m\}$$
$$Link_i(as_{jk}) \equiv \bigcup_{m=1}^{|c_j|} \{o_{jm} \mapsto [[[[Z_i]_2]_j]_k]_m\}$$

where $[P]_n$ is the n-th element in the tuple PD

At this point, we eliminate from OG the graphs which do not satisfy the invariant Inv, or update OG as follows:

$$OG = \{G_i \in OG |\ \forall o_{jk} \in O.\ Eval_i[o](Inv(c_j))\}$$

The mapping $Eval_i[o] : Exp \to V$ relates an OCL expression to the value which is obtained by evaluating the expression in the context of the object o in the graph G_i. The set V is the value set of all the OCL expressions and defined as $V \equiv Int \cup Bool \cup O \cup 2^O$.

2) Composition of statechart diagrams: Firstly, we compute the product of arguments for each action $a \in A$:

$$W_a \equiv \prod_{j=1}^{ArgN(a)} ArgD(a, j)$$

For the argument tuple $w \in W_a$, $[w]_n$ represents its n-th argument.

Then, we define the LTS for the object graph $G_i \equiv (O, Val_i, Link_i)$ as follows:

$$E_i = (P_i, cinit_i, Q_i, R_i)$$
$$P_i : 2^{O \to S},\ \ cinit_i \in P_i,\ \ Q_i \subset A,$$
$$R_i : P_i \times P_i \times Q_i \times W,\ \ W \equiv \bigcup_{a \in A} W_a$$

The set P_i is the set of states. Each state is represented by the mapping from an object to its state, i.e, if $p(o) = s$ for the state $p \in P_i$, the object o is in the state $s \in S$. The state $cinit$ is the initial state. The set Q_i is the set of actions. The set R_i

is the set of transitions. For the transition $(p, q, a, w) \in R_i$, the sets p and q are the source and destination states, a is the action to trigger this transition and w is the argument tuple of the action a.

Now, we generate each LTS E_i by the following algorithm. For simplicity, we present the algorithm for the case where the OCL expression of a synchronous transition always evaluates to a single object.

1) Let $P_i = Q_i = R_i = \phi$.
2) Define the initial state $cinit_i$ so that $cinit_i(o_{jk}) = init_j$.
3) Let $p = cinit_i$ and $P_i = \{u\}$ (p is a variable for temporal use).
4) For each object o_{jk}, transition $(p(o_{jk}), c, a, s, st) \in T_j$, and argument $w \in W_a$, do the following steps.
5) If the guard condition $Eval_i[o_{jk}][p][w](c) = true \in Bool$, create a new state q with $q(o_{mn})$ defined as follows:
   - If $m = j$ and $n = k$, then s.
   - If there exists a synchronous transition $(x, t_1, t_2) \in st$ such that the target object $Eval_i[o_{jk}][p][w](x) = \{o_{mn}\}$ and $p(o_{mn}) = t_1$, then t_2.
   - Otherwise, $p(o_{mn})$.

   Here, the mapping $Eval_i[o][p][w] : Exp \rightarrow V$ relates an OCL expression to the value which is obtained by evaluating the expression in the graph G_i and the state p, with the self object o and the action argument w.
6) Let $Q_i = \{a\} \cup Q_i$ and $R_i = \{(p, q, a, w)\} \cup R_i$.
7) If $q \in P_i$, continue the current loop. Otherwise, let $P_i = \{q\} \cup P_i$ and $p = q$, and go to 4.

V. Environment generator

We implemented the environment generator which inputs the environment model as a text file and outputs the environment instances as Promela scripts. It is used in the command line as follows:

```
% envgen rtos.env
104 cases are generated (188 milli-secs).
% cd rtos_cases; ls
case1.spin case2.spin case3.spin ...
```

The command `envgen` inputs the environment model `rtos.env` and outputs the instances `case`n`.spin` under the directly `rtos_cases`. If the target system is implemented as a file `rtos.spin`, it can be verified by placing it in `rtos_cases` and doing, for example, `spin -a case1.spin; gcc pan.c; ./a.out`.

Let us explain about the interface between the target system and the environment model. Basically, the interface matches if the script of the target system contains the functions which are defined in the target class of the environment model. For example, for the driver function `ActivateTask(tid)`, there must be the function `ActivateTask(tid)` in target system. For reference functions, we need to add an extra argument to express their return values. For example, the reference function `GetTaskState(tid)` must be implemented as the function `GetTaskState(tid,ret)` in target system. This is because the function in Promela is "inline". To express a return value for an inline function, we need to pass a variable to its argument and let the function set the return value to the variable. So, the function `GetTaskState(tid,ret)` must be implemented so that it sets the return value to `ret` in its body, e.g., `ret=tsk[tid].stat;`.

```
Rdy_Run_Fre:
  get_RTOS_info();
  assert
  (ret_GetTaskState_1==ret_GetReady &&
   ret_GetTurn==2);
  if
  :: TerminateTask(2) -> goto Run_Sus_Fre;
  :: GetResource(2,1) -> goto Rdy_Run_Occ2;
  fi;
```

Figure 6. Promela script for the environment (partially)

With this interface rule in mind, let us see how the LTS states are translated into Promela. Fig.6 shows the Promela script for the state `Rdy_Run_Fre` of the LTS in Fig.5. Firstly, inside the inline function `get_RTOS_info()`, the inline functions corresponding to the reference functions are called. For example, `GetTaskState(1,ret_GetTaskState_1);` is called. The second argument represents the return value of the reference function `GetTaskState(1)`. Then, this return variable is used in `assert`. The assertion in the environment model is translated into Promela by replacing the calls of the reference functions with the corresponding return variables. Other expressions in the assertion such as `self@Run` and `self.id` are evaluated in the translation process. Finally, driver functions are called non-deterministically to transit to the destination states.

VI. Experiment

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 following the approach in [12]. We call this model *RTOS model*. For the verification, we constructed the environment model based on the specification. We have presented this model in the examples so far. We generated the environment instances by the tool, and conducted model checking for some of the environments in Spin.

Fig. 7 shows the number of generated environments and the time taken for the generation. The computation time increases exponentially with the number of tasks and resources. But, we can generate environments efficiently

R/T	1	2	3	4
0	4 (0.0s)	20(0.0s)	140 (0.5s)	1540 (81.3s)
1	8 (0.0s)	104 (0.2s)	1496 (31.6s)	30664 (9.4h)
2	12 (0.1s)	468 (2.6s)	15132 (56.1m)	N/A
3	16 (0.1s)	1840 (61.4s)	N/A	N/A

Figure 7. The number of generated environments (CPU:2.4GHz, Memory:4.0GB)

for the cases which capture the important properties of the RTOS. Some of the properties are:

1) The task of the high priority must be executed before that of the low priority.
2) If the task of the low priority occupies a resource whose priority is higher than the task of the high priority, it is executed before the task of the high priority.
3) If a task occupies multiple resources, the priority of the task is risen to the maximum priority of the resources.

For checking the property (1), we need at least 2 tasks. For (2), we need at least 2 tasks and 1 resource. For (3), we need at least 1 task and 2 resources. In any case, we can generate the environments in a small amount of time. Of course, we need to verify the cases with more objects to increase the reliability of the properties. Furthermore, we need to verify for a wider range of variation for the task priority and resource priority (This case is only for the ranges {1,3} and {2,4}). To do this, we need a technique to improve the computation time of the environment generation (See 7.2).

We checked some of the environments in Spin. The result is that we could not find any errors in the cases of less than 3 tasks, but found errors for some of the cases of more than 2 tasks. Specifically, assertion errors (state inconsistencies) occur in the cases where multiple tasks of the same priority can become ready. When the tasks of the same priority become ready, the RTOS model is designed to run the task which became ready first. This is realized by remembering their order in the ready queue. On the other hand, the environment model is defined so that it chooses one of these tasks non-deterministically as we explained in 3.1. So, it contains the transition in which the order of execution is reversed, i.e., a task is made to run before the one which became ready before the task. When this transition occurs, the states between the RTOS model and the check model become inconsistent. This error is a false negative and not that of the RTOS model. False negatives can happen because the environment is modeled by over-approximating the ideal behavior of the environment (See 7.1).

VII. Discussion

A. Expressiveness of the environment model

The environment model expresses the state transitions in OCL. This expressiveness is not always enough to describe the accurate behavior of the environment. For example, as we explained in 6.2, RTOS chooses the task to run based on the ready queue, but the environment model cannot express such execution history. One could think of introducing the ready queue also in the environment model. But this makes the complexity of the environment model same as that of the RTOS model. This results in the need for verifying the environment model itself. Furthermore, the ready queue is an internal object of the RTOS, and not considered an environment. So, we kept the environment model as simple as possible by approximating the ideal behavior with non-determinism. Under this approximation, false positives can occur as explained in the experiment since the model contains the sequences which cannot occur in the RTOS model. Currently, we are excluding the false negative cases structurally using invariants. For example, the above case can be excluded by the invariant: "If there are more than 2 tasks, their priorities are different from each other." In order to check more precise properties such as "The RTOS schedules the ready tasks of the same priority in the order of FIFO", we need to construct by hand the specific environments which can capture the property.

B. Parallelizing model checking

To make our method effective, we have to address the problem: "How can we efficiently check all the generated environments?" As the number of the generated environments becomes quite large, it is unrealistic for users to check all of them by hand. Even if we check them automatically using some script languages, it will require a lot of time. This problem can be solved by taking advantage of the fact that each case can be checked independently of others. Specifically, we can check all the cases in parallel by distributing them to PC clusters. For example, if we distribute them uniformly to 1000 PCs, we can reduce the time simplely to 1/1000. Along with the distribution, we need to consider an effective way to feedback the check results to the user. This is a problem of data mining, i.e., how to retrieve a useful information from the huge number of check results. For this problem, we consider it effective to display all the results as a list of Boolean values indicating if each case contains an error or not. This allows us to have a bird's eye view of all the results to identify the boundary of the error cases such as "The cases with less than 3 tasks are OK, but not for more." We also need to devise the user interface to navigate from the list to the details of the error messages.

We can also parallelize the generation of environments. As we mentioned, we are generating environment as a stream. By breaking up this stream into fragments and distributing them to PC clusters, we can reduce the computation time of the generation. For example, the time to generate the case of 4 tasks and 1 resource, which took more than 9 hours by a single PC, would be reduced to less than 1 minute by 1000 PCs.

VIII. Related work

O. Tkachuk, et.al [6] proposes Bandera Environment Generator (BEG) which automatically generates the environment for the verification of Java programs in Bandera. 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. On the other hand, our method can express the set of the instances as a class model and automatically generate possible instances based on the variations of the class model.

J. Penix, et.al [8] verifies the time partitioning of DEOS RTOS by Spin. The environment is obtained from the universal environment which tries nondeterministic function calls to the target system by constraining it with the assumption described in LTL [9]. This method is effective when the assumption can be described simply, but this becomes difficult when the environment includes complex transitions. In our method, we do not consider the universal environment, but describe the specific range of function calls using statechart diagrams. This allows us to describe a more precise behavior of the environment. Our method also deals with the variation of the environment structure.

P. Parizek, et.al [7] proposes a method to combine the model checkers Java PathFinder and Protocol Checker. This method targets at the verification of Java components whose protocols are described in ADL (Architecture Description Language). It conducts model checking by searching the program states by Java PathFinder in the Java part and by Protocol Checker in the ADL part. Although the environment can be modeled in ADL, it cannot express the variation of the environment using classes like in our method.

J. Lilius, et.al [11] proposes vUML for verifying UML models in Spin. It verifies statechart diagrams by translating into Promela and feedbacks the error trace as sequence diagrams. Our method is similar to this work in that it deals with the statechart diagrams. But we are using them for describing the environment, not the target system itself. As the environment describes the specification which must be met by the target system, it should be described declaratively. So, we introduced the pre- and post-conditions style notation instead of the standard event communication. vUML also has the facility to create environment, but it only inputs the external events non-deterministically to the system.

IX. Conclusion

In this paper, we presented a UML-based method for modeling and generating the environments for the model checking of embedded systems. We also presented a tool to automatically generate Promela/Spin scripts from the environment model. Using the tool, we conducted the verification experiment of an OSEK/VDX RTOS design model. The effectiveness of our method is summarized as follows: (1) The model can be described in UML which is familiar to most of the engineers. (2) It avoids the state explosion problem by dividing the whole environment into small cases based on its structure. (3) It can be generally applied to the verification of reactive systems especially for those whose environment has a lot of structural variation such as OS and middleware.

Future work is, on the theoretical side, to prove the correctness of the environment generation algorithm, and on the practical side, to implement a parallel distributed model checking framework based on our method.

References

[1] OMG. Unified Modeling Language. URL: http://www.omg.org/.

[2] J. Warmer and A. Kleppe. The object constraint language: precise modeling with UML. Addison-Wesley, 1999.

[3] G.J.Holzmann: The Spin Model Checker - Primer and Reference Manual, Addison-Wesley, 2004.

[4] Jeff Magee and Jeff Kramer: Concurrency: State models & Java programs, , 1999.

[5] OSEK/VDX, URL:http://portal.osek-vdx.org/.

[6] O. Tkachuk, et.al: Automated environment generation for software model checking. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003.

[7] Pavel Parizek and Frantisek Plasil: Specification and Generation of Environment for Model Checking of Software Components, Electronic Notes in TCS, pp.143-154, 2007.

[8] John Penix, Willem Visser, et.al.: Verification of Time Partitioning in the DEOS Scheduler Kernel, International Conference on Software Engineering, pp.488-497, 2000 .

[9] Corina S. Pasareanu: DEOS Kernel: Environment Modeling using LTL Assumptions, NASA Ames Technical Report NASA-ARC-IC-2000-196, 2000.

[10] Oksana Tkachuk and Sreeranga P. Rajan :Application of automated environment generation to commercial software, International Symposium on Software Testing and Analysis, 2006.

[11] J. Lilius and I. Paltor. vUML: A tool for verifying UML models. In Proceedings of Automated Software Engineering, ASEf99. IEEE, 1999.

[12] Toshiaki Aoki, Model Checking Multi-task Software on Real-time Operating Systems, International Symposium on Object-Oriented Real-Time Distributed Computing 2008, pp.551-555, 2008.