Combined Model Checking and Testing Create Confidence in Correctness of Commercial Automotive Operating System

Toshiaki Aoki
School of Information Science,
Japan Advanced Institute of Science and Technology

Makoto Satoh
Renesas System Design Co., Ltd.

Mitsuhiro Tani
DENSO CORPORATION

Kenro Yatake
School of Information Science,
Japan Advanced Institute of Science and Technology

Tomoji Kishi
Faculty of Science and Engineering
Waseda University

May 23th, 2016.
IS-RR-2016-002

Combined Model Checking and Testing Create Confidence in Correctness of Commercial Automotive Operating System

Toshiaki Aoki
School of Information Science
Japan Advanced Institute of Science and Technology
Email:toshiaki@jaist.ac.jp

Makoto Satoh
Renesas System Design Co., Ltd.
Email:makoto.sato.jz@renesas.com

Mitsuhiro Tani
DENSO CORPORATION
Email:tani@eeda.denso.co.jp

Kenro Yatake
School of Information Science
Japan Advanced Institute of Science and Technology
Email:k-yatake@jaist.ac.jp

Tomoji Kishi
Faculty of Science and Engineering
Waseda University
Email:kishi@waseda.jp

***Abstract*—The safety and reliability of automotive systems are becoming a big concern in our daily life. Recently, a functional safety standard which specializes in automotive systems has been proposed by the ISO. In addition, electrical throttle systems have been inspected by NHTSA and NASA due to the unintended acceleration problems of Toyota's cars. In light of such recent circumstances, we are researching practical applications of formal methods to ensure the high quality of automotive operating systems. An operating system which we focus on is the one conforming to the OSEK/VDX standard. This paper shows a case study where model checking is applied to a commercial automotive operating system. In this case study, the model checking is combined with testing in order to efficiently and effectively verify it. As a result, we acquired the confidence that the quality of the operating system is very high.**

Keywords: automotive operating systems, model checking, testing, design verification, test case generation

I. Introduction

Recently, the safety and reliability of automotive systems are becoming a large concern in society. Although vehicles have been controlled by simple mechanics in the past, many of electronic parts are embedded in them at present according to the progress of electronic control technology and its performance. These electronic parts can actualize the complex control of the vehicles, and make it possible to provide high functionality to vehicles such as automatic speed controlling and emergency braking. The electronic control technology makes the vehicles more convenient and safer. Unfortunately, electronic parts also introduce the problems of the reliability and safety of the vehicles because the automotive systems become more complicated and their scale larger. In fact, highly electronized automotive systems have received much attention with respect to their reliability and safety. A functional safety standard which specializes in automotive systems has been proposed by the ISO[4]. Electronic throttle systems have been inspected by NHTSA and NASA because of the unintended acceleration problem of Toyota's cars in 2010[1].

We are working on the verification of automotive operating systems to ensure the high quality of automotive operating systems. An operating system which we focus on is the one conforming to the OSEK/VDX[2] standard. OSEK/VDX is an organization which was established in 1993 and provides the industrial standards of ECU(Electronic Control Unit) architectures. OSEK/VDX deals with many kinds of components used in automotive systems and one of them is an operating system. Although AUTOSAR[3] takes over this activity, the OSEK/VDX standards is still used for automotive operating systems in practice. We use OS for the abbreviation of 'operating system' below.

Our purpose is to provide a high quality OS by applying formal methods which are recommended in the functional safety standards. OS has much impact on their safety evaluation because it is the base of automotive software which is embedded into automotive systems. JAIST and DENSO started a joint research project in 2006. DENSO develops automotive software using OSs which are provided by the other companies. We examined the feasibility of applications of formal methods at this point. Then, we decided to apply formal methods to a commercial OS whose target CPU is V850. Renesas Electronics Corporation(REL) which develops this OS and CPU joined this project in 2009. We call the OS 'REL OS' below. REL OS has been already released and used in a current series of cars at this time. It is needless to say that traditional methods have been applied to REL OS in order to check it then. Our aim is to achieve higher quality of the OS for next series of cars by applying formal methods.

This paper shows a case study that model checking, which is one of formal methods, is applied to a commercial OS, that is, REL OS. REL OS is too complicated to convince us that it correctly performs for any application. We adopted exhaustive verification techniques to check REL OS. We have conducted exhaustive testing based on a design model which was exhaustively verified by model checking. As a result, we acquired the confidence that REL OS correctly performs for any application although no new bug was found since the model checking and testing were more exhaustive and reliable than the traditional methods. Such combined model checking and testing are appropriate to convince us of the correctness thanks to their exhaustive nature.

The rest of the paper is organized as follows. We briefly introduce OSEK/VDX OSs in Section 2. Section 3 shows the overview of our approach to apply model checking and testing to the verification of REL OS. Section 4 discusses related works. Section 5 and Section 6 explain the details and results of our application. Section 7 discusses the approach and results. Section 8 concludes this paper.

II. OSEK/VDX Operating Systems

OSEK/VDX OS, shortly, OSEK OS adopted a priority based scheduling of multiple tasks. Mixing preemptive tasks with non-preemptive tasks is allowed. It provides API functions such as ActivateTask, TerminateTask, and ChainTask for controlling the execution of tasks. ActivateTask activates a task, TerminateTask terminates a task, and ChainTask activates a tasks after terminating a task. A concept named resource exists to manage shared resources. The resource is obtained and released by API functions GetResouce and ReleaseResource respectively. Mutual exclusion of tasks which have access to the shared resource can be realized by these API functions. They adopt a priority ceiling protocol[6] in order to avoid a priority inversion problem. Interrupt service routines are invoked when the interrupts occurred. We abbreviate interrupt service routines as ISR below. A priority is assigned to an ISR like a task. Synchronizing the tasks by events and alarms which invoke tasks based on time are also provided in addition to the API functions shown in the above.

The primary function of OSEK OS is to schedule tasks and ISRs. It is not too much to say that OSEK OS is almost a scheduler. In the scheduler, information needed for the scheduling is managed by data structures such as a queue and tables. The scheduler determines a task or ISR to be executed by computing with those data structures. Such computation is very complicated since there are various configurations of priorities and preemptions, activation timings of tasks, and ISRs, and synchronization mechanisms. It is very important to ensure that the computation is correct for any configuration. How the scheduling has to work is defined by OSEK/VDX standard specification. In our joint research project, we verified the fact that the scheduler of REL OS surely conforms to the specification.

III. Approach

We show the overview of our approach in Figure 1. Our approach is divided into two kinds of activities, design verification and testing. We have constructed a design model to clarify computation carried out in REL OS. We confirmed that the computation is correct by applying model checking to the design model. Incorrect scheduling as shown in Section 2 would be detected at this point. Then, we have conducted testing based on the design model in order to confirm that the implementation of REL OS conforms to the design model in which the correct computation was realized. The testing provides us the confidence that the implementation is correct because it is actually performed. The activities associated with the design verification and the testing are surrounded by solid lines and dotted lines respectively.

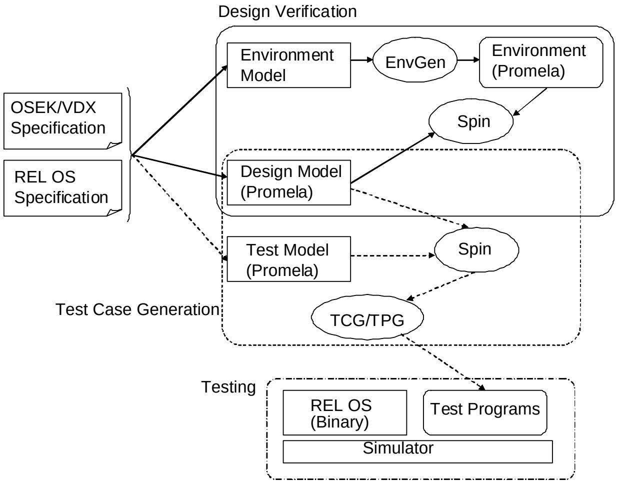


Fig. 1. Overview of Approach

A. Design Model

We constructed and verified a design model of REL OS to analyze its scheduling mechanisms. The design model was verified by a model checking tool Spin[7]. Spin checks properties represented as LTL formulas, assertions and so on against behavior represented as automata which are concurrently executed with channels to communicate with each other. Such behavior is described in a specification language named Promela. A Promela description consists of concurrent processes whose behavior is described as guarded commands in an operational way. It also provides various datatypes such as arrays and record types. Those datatypes allow us to straightforwardly describe the scheduling mechanisms of REL OS. In addition, the syntax of Promela is similar to C language which is familiar with engineers who develop REL OS. It is easy for the engineers to learn as well as communicate with researchers based on Promela descriptions. Thus, we constructed the design model in Promela.

B. Design Verification and Environment Modelling

1) Closing Open System By Environments: OSEK OS is an open system which performs if an API function is invoked by tasks and ISRs. It does not do anything if nothing is invoked. Similarly, the design model is not executable by itself. To check it by Spin, we need descriptions making invocations of API functions in addition to the design model. Such descriptions are usually called *environments* because it is outside of the design model.

2) Types of Environments: The environments deal with invocations of functions to a target, callback from the target, inputs to the target and outputs from the target. There are two types of the environments. One is that the environments make completely non-deterministic invocations of functions and inputs to the target. The other is that they make non-deterministic invocations within specific execution contexts. The former is called *universal environments*[22]. Although the universal environments allow us to exhaustively check behavior of the target, many spurious errors will be reported. To avoid them, we need to constrain the behavior of the universal environment, for example, provide a filter of the behavior by the form of LTL formulas [10]. Our objective of the design verification is to ensure the fact that a task selected by the scheduler is correct with respect to the specification. To define this fact, we need to constrain the behavior of the environment so that it can make invocation sequences which lead to the selection of a specific task. However, it is very hard to describe LTL formulas representing those invocation sequences. In addition, the universal environment likely causes a state explosion problem. Thus, in our approach, we adopted the latter type of the environments, which make non-deterministic invocations within specific execution contexts. Those environments are described in automata which define the contexts. They make it easier to describe the invocation sequences as the environments.

3) Facilitating Variations of Environments: In our approach, the environments are described as an environment model that we have proposed in [13]. There are a number of variations of configurations such as the number of tasks, the number of resources, priorities of tasks, and ceiling priorities of resources for the environments. It is very hard to manually describe those environments one by one. Thus, we have proposed the environment model which represents variations of the configurations and allows us to automatically generate environments described in Promela. The variations are modeled in the class diagram of UML[11] and OCL(Object Constraint Language)[12]. The invocation sequences of API functions are modeled in the statechart diagram of UML with some extension. The expected results of the invocation of the API functions are also described in the statechart diagram with OCL. Then, we have developed a tool named *EnvGen*[13], [14] which automatically generates environments described in Promela from the environment model consisting of the class diagram and statechart diagram. The environments are generated within specific bounds of the variations of the configurations. The expected results are realized as assertions in them.

4) Constructing Environment Models: The possible configurations of the environments of REL OS are described in the environment model. Such configurations are identified from the specification of REL OS in addition to the OSEK/VDX standard specification. The invocation sequences and expected results of the API functions are also identified and described in the environment model similarly. In verifying the design model of REL OS, we generated environments from this environment model. The design model is coupled with each of the environments generated by EnvGen, then it is checked by Spin. That is, the design model is checked as many times as the number of the environments.

C. Testing

1) Ensuring Conformance by Testing: We made much effort to ensure that the scheduling of tasks was correctly realized in the design model. What we had to do next was to ensure that the implementation of REL OS conforms to the design model. There are two approaches to ensure that an implementation conforms to a design model in general. One is that we generate a source code from the design model. The other is to check that the implementation conforms to the design model after manual implementation. We selected the second approach. A primary reason why we selected this approach was that REL OS has been already implemented. Another reason was that it was very hard to refine the design model so that a source code can be generated. REL OS is implemented in an assembly language of V850 to achieve high performance of executions. In addition, there are many mechanisms and optimizations, which are specific to V850. However, some such mechanisms do not appear in the design model since it focuses on the computation of the scheduling. Therefore, it would be very hard to refine the design model so that it can be isomorphic to the implementation of REL OS.

2) Regarding Design Model as Test Oracle: In our approach, we test the implementation of REL OS by test cases which are generated from the design model in order to check that the implementation conforms to the design model. We assigned much importance to the verification of the design model. We not only checked the design model by Spin but also reviewed the design model and environment model carefully. As the result of this effort, we put the assumption that the design model is correct, that is, regard it as a test oracle. This assumption is reasonable since the design model must be relatively reliable in comparison to the other artifacts. Test cases are generated from the design model. Those test cases contain invocations of API functions in addition to expected results of them. Obtaining the expected results is possible because correct computation (which is the assumption) is done in the design model.

3) Covering Implementation States: Conformance testing based on automata has been studied for a long time[16]. By these studies, it is well-known that ideal assumptions are needed to decide that one automata conforms to another one. However, it is difficult for practical systems to discharge those assumptions. Therefore, we do not aim at this theoretical conformance but cover all the states, which appeared in the design model. In this approach, in order to cover states that we expect to test in the implementation, the design model needs to contain corresponding states. Thus, we constructed the design model so that if states of the implementation are different to each other, corresponding states of the design model can be also different to each other. This makes it possible to generate test cases, which reach expected states of the implementation.

We use a model checking tool to obtain test cases which cover all the states of the design model.

4) Environments for Test Case Generation: We need environments in generating test cases as well as in the design verification since the design model does not perform if no API function is invoked. We call the environments to generate test cases *test models*. The test models are different from the environments of the design verification. The test models do not check the design model but only invoke API functions non-deterministically within some bounds.

5) Tools to Automate Testing: We have developed two tools named TCG and TPG for automatic testing based on the design model. TCG automatically generates test cases using Spin. Our approach is to generate test cases not by trap properties[18] but exhaustive search algorithm of states with a model checking tool. TCG generates test cases, which are reachable to all the states appearing in the combination of the design model and test models. Generated test cases consist of invocation sequences of the API functions and expected results. TPG transforms the test cases into programs to test REL OS. A program generated by TPG is compiled with REL OS and executed in a simulator and debugger of V850. TCG and TPG allow us to automatically perform testing of REL OS using the design model and test models as inputs.

IV. Related Works

A word 'verification' is recognized as proving correctness with theorem provers or deductive techniques. The verification of OSs is challenging as demonstrated by the existing researches[19]. The verification of seL4 kernel is known as a recent notable success story[20], [21]. The word 'verification' is not limited to such deduction based approaches but used for model checking ones. J. Penix, et al.[22], [9] verifies the time partitioning of DEOS. In this work, 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 weaknesses when describing precise behavior of environments due to the accumulation of complex LTL assumptions. In our approach, we adopted different types of environments using automata to simply describe properties of scheduling. In addition, this work only verifies the design model despite that our approach deals with not only the verification of the design model but also the testing of the implementation.

There are several works on the verification of OSEK OS. L.Zhu, et.al[23] verifies OSEK OS implemented in C language. The primary purpose of this work is to formally specify API functions of the OSEK OS. A part of such specifications is verified by VCC[25]. Y.Huang, et.al[24] manually construct a model of CSP based on the source code of OSEK OS. Then, it is checked by a model checking tool PAT[26]. They do not take the conformance between the model and the source code into account. Y.Choi[27], [28] verifies an open source OS named Trampoline[29], which is implemented in C language. In this work, the source code of Trampoline is analyzed by Spin. A model of Promela is manually constructed, then it is checked against properties obtained by safety analysis. This work does not take the conformance between the model and the source code into account as well. In comparison to those works, the originalities of our work can be summarized as follows. Firstly, our approach covers both of design and testing phases of developments although the other works focus on a single activity or phase of the developments. We combine the verification of the design model with testing of the implementation seamlessly. Secondly, our target is implemented in the assembly language of V850. Thus, we cannot take existing techniques which are specific to C language like [27], [28] and [23].

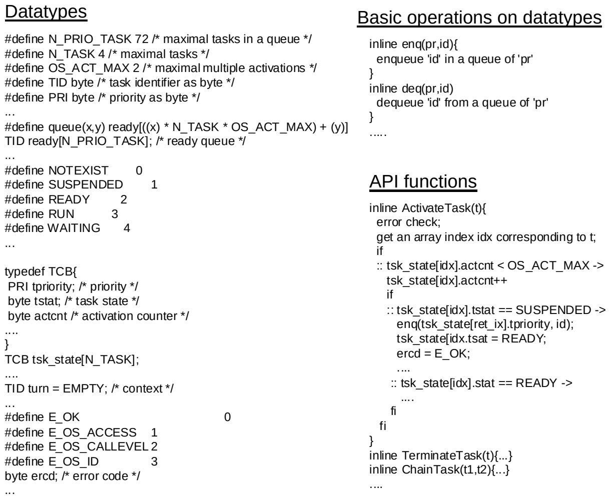
In our previous works, we have proposed a tool to automatically generate environments [13] and it has been applied to a design model of OSEK OS[15]. We adopt the tool for verifying a design model of REL OS. The design model described in [13], [15] is different from that of REL OS. We have proposed an approach to automatically generate test cases from the design model[17]. In the approach, test scenarios to generate the test cases were described in Z notation[30]. We do not describe the test scenarios in this paper. Instead of the test scenarios, we describe a test model which non-deteministically invokes API functions of OSEK OS to exhaustively generate test cases since our purpose of the verification is to obtain the confidence thanks to the exhaustive nature. In addition, we made some trade-offs and decisions for obtaining the confidence throughout the design verification and testing. In this paper, we show a practically integrated approach to obtain the confidence and experiences that we gained in the case study.

V. Design Model and Verification

A. Construction of Design Model

We constructed the design model of REL OS in Promela. As mentioned in Section 2, we focus on its scheduler. The scheduler of REL OS has data structures consisting of a ready queue, tables, and flags. The ready queue records activation orders of tasks for each of the priorities. A task to be executed is determined based on the ready queue. It is obtained by searching the highest and firstly activated task recorded in the ready queue. The tables record information of tasks such as the current states and priority of tasks. The flags record conditions needed for the scheduling. High performance of the scheduling is required for the OS since it controls machinery of automobiles. On the other hand, searching the ready queue and switching tasks are costly in time. To achieve high performance, the flags are used for identifying whether costly operations are needed or not. Such data structures can be straightforwardly described in Promela.

Figure 2 shows a part of the design model described in Promela. In the implementation of REL OS, the ready queue is realized as a specific memory area of ECU. Operations to enqueue/dequeue a task to/from the ready queue is implemented by instructions of V850 which compute addresses to update the memory area. We did not model the ready queue based on such memory area and instructions. We modeled the ready queue as an array instead of the address computation. In Figure



Datatypes

```
#define N_PRIO_TASK 72 /* maximal tasks in a queue */
#define N_TASK 4 /* maximal tasks */
#define OS_ACT_MAX 2 /* maximal multiple activations */
#define TID byte /* task identifier as byte */
#define PRI byte /* priority as byte */
...
#define queue(x,y) ready[((x) * N_TASK * OS_ACT_MAX) + (y)]
TID ready[N_PRIO_TASK]; /* ready queue */
...
#define NOTEXIST       0
#define SUSPENDED       1
#define READY        2
#define RUN          3
#define WAITING      4
...

typedef TCB{
 PRI tpriority; /* priority */
 byte tstat; /* task state */
 byte actcnt /* activation counter */
....
}
TCB tsk_state[N_TASK];
....
TID turn = EMPTY; /* context */
...
#define E_OK                          0
#define E_OS_ACCESS    1
#define E_OS_CALLEVEL 2
#define E_OS_ID             3
byte ercd; /* error code */
...
```

Basic operations on datatypes

```
inline enq(pr,id){
  enqueue 'id' in a queue of 'pr'
}
inline deq(pr,id)
  dequeue 'id' from a queue of 'pr'
}
.....
```

API functions

```
inline ActivateTask(t){
  error check;
  get an array index idx corresponding to t;
  if
  :: tsk_state[idx].actcnt < OS_ACT_MAX ->
     tsk_state[idx].actcnt++
     if
     :: tsk_state[idx].tstat == SUSPENDED ->
        enq(tsk_state[ret_ix].tpriority, id);
        tsk_state[idx.tsat = READY;
        ercd = E_OK;
        ....
      :: tsk_state[idx].stat == READY ->
         ....
      fi
   fi
}
inline TerminateTask(t){...}
inline ChainTask(t1,t2){...}
....
```

Fig. 2. Design Model

2, the ready queue is represented by an array named 'ready' in the design model. Operations to enqueue and dequeue are described as inline macros which update it. TCB (Task Control Blocks) which hold information of tasks is represented by an array named 'tsk_state' as well as the ready queue.

If an API function is invoked, a task to be executed is determined after values of the datatypes are updated. For example, if ActivateTask(t) where t is a task identifier is invoked, t is enqueued into the corresponding position of the ready queue as well as a current state of a task t which is recorded in the tables is updated to a ready state. Then, a search of the ready queue and switching a task are performed if needed. Such operations are realized as inline macros of Promela in the design model as shown in Figure2. Interrupt handling mechanisms are described in the design model in addition to operations of API functions. The interrupts affect the scheduling of tasks although they are processed by hardware in actuality.

The design model has been constructed by the JAIST side in our joint project. The initial design model has been constructed based on the OSEK/VDX standard. However, its behavior was not the same to REL OS due to some misunderstandings and the ambiguity of the specification. It was reviewed by engineers of DENSO and REL to make its behavior equivalent to that of REL OS. We held regular meetings to consider review results once a month. The improvements of the design model were also done by the JAIST side according to the review results. It took around six months to improve it and we finally obtained the design model of REL OS.

B. Approximation of Environments

As mentioned in Section 2, in our approach, the environments are described as an environment model that we have proposed. We describe our decisions and tradeoffs that we made with brief introduction of the environment model here. For more details, please refer to our earlier works [13], [14], [15]. The environment model represents the set of invocation sequences of API functions and their expected results. They are analogous to test cases and their expected results in a sense. We want to examine the design model ultimately for all the invocation sequences. However, if we construct the environment model which deals with those invocation sequences, its complexity becomes similar to that of the design model. For example, if we describe expected results in the case where multiple tasks whose priorities are the same are activated, we need a queue like the ready queue of the design model in the environment model because activation orders of tasks have to be recorded in it as well. Creating such an environment model makes little sense because its reliability becomes as uncertain as the design model. The reliability of the environment model should be higher than the design model from the viewpoint of practicality.

There are two approaches, over approximation and under approximation, to solve this problem. In the former, we construct an environment model so that it can contain more invocation sequences and stronger expected results than exact ones. For example, if there are multiple tasks whose priorities are the same and in the ready states, an expected result is that one of those tasks should be executed. In this case, we do not need a queue to describe expected results and make the environment model simpler. However, such expected results make little sense because they are too strong. That is, for example, the fact that one of tasks whose states are ready should be running does not contribute to the correctness of the scheduling very much. In addition, many false positive counter examples would be reported because the environment model contains many of non-deterministic invocations of API functions. Hence, we took the latter approach which is under approximation. In this approach, we construct an environment model so that it can contain less invocation sequences and weaker expected results than exact ones. This under approximation is achieved by case splittings of environment models. Thus, we constructed the environment models separately such that each model was as simple as possible. This separation allows us to provide high reliability to them. In addition, the separation of models can reduce the risk of state explosion. If we check all aspects of the design model at once, state explosion can easily occur. We can check each of them within a relatively small state space by separating the environment models.

C. Verification Result

We construct environment models separately. Such separation introduces a problem of coverage. This approach does not cover all the invocation sequences of API functions because it is based on the under approximation. On the other hand, as discussed so far, neither fully non-deterministic nor over approximated invocations of API function can be accepted in our approach. Thus, we decided to take the under approximation approach and carefully separate the environment models so that they can cover our concerns in the design model.

No.	Name	Purpose	Condition
1	TaskDiff	termination and activation of tasks	different priorities
2	TaskEq	termination and activation of tasks	same priorities
3	CtDiff	ChainTask	different priorities
4	CtEq	ChainTask	same priorities
5	MultDiff	multiple activation of tasks	different priorities
6	MultEq	multiple activation of tasks	same priorities
7	ResDiff	get and release of resources	different priorities
8	ResEq	get and release of resources	same priorities
9	EvDiff	events	different priorities
10	EvEq	events	same priorities
11	IsrDiff	ISR	different priorities
12	IsrEq	ISR	same priorities

Fig. 3. Separated Environment Models

No.	name	generated environments					model checking
		num.	time(s)	lines	states	trans.	time(s)
1	TaskDiff	26	0.6	153	4	9	115.6
2	TaskEq	9	0.3	245	9	16	44.2
3	CtDiff	26	0.8	168	4	13	116.3
4	CtEq	9	0.3	273	9	23	45.9
5	MultDiff	26	0.9	199	8	19	119.7
6	MultEq	9	1.1	576	50	98	68.3
7	ResDiff	341	62.7	892	44	112	3828.5
8	ResEq	63	10.9	926	44	112	834.4
9	EvDiff	26	1.4	336	15	47	143.3
10	EvEq	9	1.6	1284	78	261	179.7
11	IsrDiff	182	13.9	502	17	49	1245.0
12	IsrEq	63	5.9	789	34	99	617.0

Fig. 4. Environment Generation and Model Checking Results

Figure 3 shows environment models that we have constructed. The environment models are divided into six groups based on which the functionalities of OSEK 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 task management functions. They represent the cases with different priorities and the same priority respectively.

Figure 4 shows the results of environment generation and model checking. We used a computer whose specification is Intel Core2Duo CPU 2.4GHz with 4Gbyte memory in them. 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 descriptions, 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. 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 789 environments in about 100 s, 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 design model using all of the environments without state explosion occurring due to the separation of the environment models. The entire model checking took 122 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 Promela descriptions.

We conducted model checking several times while we were constructing the design model. The results shown in Figure 4 are final ones. The design model was constructed and checked in this way by the JAIST side. We found many errors such as wrong conditions and wrong updates of data by model checking during its construction and verification. However, they were not the errors of REL OS but of the design model itself. That is, we described the wrong conditions and updates which do not appear in REL OS in constructing the design model. Finally, no error was reported by model checking. Nonetheless, we gained confidence that the design model was highly reliable. We encountered many errors detected by our approach even though they were not the errors of REL OS. It made us believe that it was powerful enough to find errors.

VI. Testing based on Design Model

A. Generation of Test Cases and Programs

We made much effort to verify the design model by review and model checking. Then, we put an assumption that it is correct and generate test cases with their expected results from the design model. We have proposed a method to automatically generate test cases from Promela descriptions by Spin [17]. In this method, search paths during model checking are recorded as search logs. Spin has an option to generate debug information such as up and down operations of depth-first search of model checking algorithm. In addition, we can print out information about the status of the design model such as invoked API functions and the current states of tasks during model checking thanks to the embedded C function of Promela. Such debug information and status make it possible to restructure a search tree of model checking in which expected results are contained. We obtain test cases, which are reachable to all the states of the design model and test models by scanning this search tree. We have developed a tool named TCG for testing REL OS according to this method. TCG inputs the design model and a test model. Then, it outputs invocation sequences of API functions and their expected results consisting of current states and priorities of tasks.

The test cases generated by TCG are not programs but the invocation sequences and expected results. Thus, we developed a tool named TPG which translates them into programs to be compiled with REL OS. The programs generated by TPG are regarded as applications executed on REL OS. The programs invoke API functions according to the test cases. In addition, they have statements to check whether status of REL OS is the same to those expected results. Such a check is realized by debugger of a V850 development environment. The results of the check is stored in a log file of testing.

TCG and TPG allow us to automate testing of REL OS based on the design model. If we give the design model with a test model, testing is automatically performed and then its results are recorded in the log file. In our project, TCG and TPG were developed by the JAIST side and REL side respectively.

B. Test Models

We need environments in generating test cases as well as the design verification since the design model does not perform if no API function is invoked. The environments to generate

```
do
:: precondition$_1$ -> API function$_1$
:: precondition$_2$ -> API function$_2$
...
:: pre-condition$_n$ -> API function$_n$
od
```

Fig. 5. Test Model

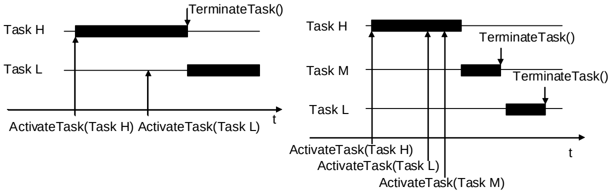


Fig. 6. Task Switches

the test cases are called test models. Figure 5 shows a general form of the test models. A test model invokes API functions of the design model non-determistically. The reason why preconditions are described is to prevent infeasible test cases from being generated. For example, invoking TerminateTask() by a task whose state is not running is infeasible in an actual execution. Such invocations are excluded by the preconditions. Execution context of the design model is needed to describe preconditions. For example, the current states of tasks are needed to describe the precondition of the invocation of TerminateTask(). This execution context can be obtained by referring to states of the design model. This is reasonable because we put an assumption that the design model is correct in testing. Checking the design model is not an objective in testing. Reference to the states of the design model make it easier to describe test models.

Configurations have to be bounded in the test models. We have to determine the number of tasks, resources, ISRs, their priority assignments and events for the generation of test cases. We investigated behavior of task switches realized by REL OS since we focus on its scheduling. We describe each of possible variations of task switches as a use case. Then, we consider that what configurations cover those variations. For example, we show two of use cases in Figure 6. Two tasks which have different priorities are sufficient to ensure the fact that a task whose priority is the highest among tasks whose states are ready. This case is shown in the left-hand side of Figure 6. To ensure the fact that a task to be executed does not depends on activation orders, we need three tasks which have different priorities. This case is shown in the right-hand side of Figure 6. In this way, we identified the following configurations which cover the variations of task switches. The numbers of tasks, resources, ISRs and events are 3,2,1, and 1 respectively.

C. Test Cases and Test Programs

Figure 7 shows a test case and test program generated by TCG and TPG respectively. The test case represents an invocation order of API functions as follows.

1) A task named task1 invokes ActivateTask(task2).
2) A task named task2 invokes ActivateTask(task3).

```
Test Case
   task1:ActivateTask(task2) task2:ActivateTask(task3) SetINTR(1)
   isr1:SetEvent(task1,Event1) ResetINTR(1)  task3:TerminateTask()
   task2:TerminateTask() task1:ActivateTask(task3) SetINTR(1)
   isr1:SetEvent(task3,Event1) isr1:ActivateTask(task2) ResetINTR(1)
   task3:TerminateTask() task2:TerminateTask() task1:ChainTask(task2)
   task2:ActivateTask(task1) task2:TerminateTask() task1:ActivateTask(task3) ...
```

```
Test Program
ISR(isr1){
 if(exccnt1 == 1){
   ercd = SetEvent(task1,Event1);
   return;
 } if(exccnt1 == 2){
   ercd = SetEvent(task3,Event1);
   ercd = ActivateTask(task2);
   return;
 }
 if(exccnt1 == 3){
 /* if branches continue */
}

TASK(task1){
  ercd = ActivateTask(task2);
  ercd = ActivateTask(task3);
  ercd = ChainTask(task2);
  return;
 }
 if(exccnt2 == 2){
  exccnt2++;
  ercd = ActivateTask(task3);
  ercd = ActivateTask(task3);
  ercd = ChainTask(task2);
  return;
  }
 if(exccnt2 == 3){
 /* if branches continue */
}

TASK(task2){
 if(exccnt3 == 1){
   exccnt3++;
   ercd = ActivateTask(task3);
   ercd = TerminateTask();
   return;
/* if branches continue */
 }
 if(exccnt3 == 2){
   ercd = TerminateTask();
   return;
 }
 if(exccnt3 == 3){
 /* if branches continue */
}

TASK(task3){
 if(exccnt4 == 1){
   exccnt4++;
   ercd = TerminateTask();
   return;
 }
 if(exccnt4 == 2){
   exccnt4++;
   ercd = SetIntr(1);
   ercd = TerminateTask();
   return;
 }
 if(exccnt4 == 3){
 /* if branches continue */
}
```

Fig. 7. Test Case and Test Program

3) An interrupt whose number is 1 occurs.
4) An interrupt service routine isr1 invokes SetEvent(task1, Event1).
5) An interrupt which occurs in 3 is reset.
6) · · ·

The expected current states such as ready queue and TCB exist in the test case, however; they are omitted here for the sake of simplicity. Timings that interrupts occur are also described. SetINTR(1) and ResetINTR(1) represent that the interrupt which triggers an interrupt service routine named isr1 is set and reset respectively.

The test program realizes the invocation order described in the test case as well as check the expected current states. Although the test program consists of tasks, the test case is an invocation sequence of API calls. Thus, we need to transform the test case to the tasks which cause the invocation sequence. The test case contains information which makes it possible to assign invocations of API functions to tasks. The invocation order of assigned API functions has to be controlled inside of each of the tasks. The variables exccnt1 to exccnt4 control the invocation order so that it can follow the one represented in the test case. The expected current states and execution order of API functions are checked using debug functions of the simulator. The timings that the interrupts occur are controlled by a library of the simulator. Code fragments to check them and control the interrupts exist in the test program, however; those are omitted in Figure 7.

D. Test Results

Figure 8 shows results to generate test cases and programs by TCG and TPG. Three tasks and two resources are named

TaskA, TaskB, TaskC, ResourceA, and ResourceB respectively. An ISR and event are omitted in this figure. Priority assignments of the tasks and resources are described in their rows. For example, upper-left of Figure 8 represents that the priorities of TaskA, TaskB, and TaskC are 1,2, and 3 respectively. Furthermore, there exist six variations of ceiling priorities of the resources. Regarding to these values, greater values mean higher priorities. Note that variations of priority assignments are reduced by their symmetry. The row of '#test cases' represents the number of test cases generated by TCG. The rows of 'pan exe. time', 'TCG exe. time', and 'TPG exe. time' represent amounts of time which are taken to search reachable states by Spin, generate test cases by TCG and generate test programs by TPG respectively where time units are seconds. We generated the test cases and test programs by a computer whose specification is Intel(R)Core2Duo CPU 3.00GHz with 1Gbyte memory. The total number of test programs generated by these configurations is 742,748.

Each of the test programs generated was compiled with REL OS and executed on the simulator of V850. These test programs can be executed independently. Thus, it is possible to perform testing in parallel theoretically. However, we used a debugger to check REL OS and the number of its licenses that we can use is limited to three. In addition, some of them are often occupied by engineers of REL. Therefore, we executed the test programs in a single computer in the daytime of weekdays, and in parallel on weekends and in the midnight of weekdays. As a result, we took around three months to complete checking all the test cases. Surprisingly, no failure of test cases was found in the testing.

We could not measure exact time taken to check all of them because testing was parallelize in an ad-hoc way. We measured time taken to execute a part of the test programs instead. For example, it took 165.75 hours to check 26,489 test programs where time to compile and execute them are 42.5 and 127.25 hours respectively. It took 265 hours to check 44,723 test programs where time to compile and execute them are 80.25 and 184.75 hours. The test programs were complied and executed by a computer whose specification is Pentium4 3.2GHz CPU with 1Gbyte memory. We can estimate the whole of time to be taken to complete the execution of the test programs based on these data as around 4,535 hours, that is, 189 days. We can say from this estimation that our parallelization (even though it is ad-hoc) contributed to the reduction of time taken for the testing because around 3 months were taken to complete it in fact.

VII. Discussion

A. Practical Applications of Model Checking

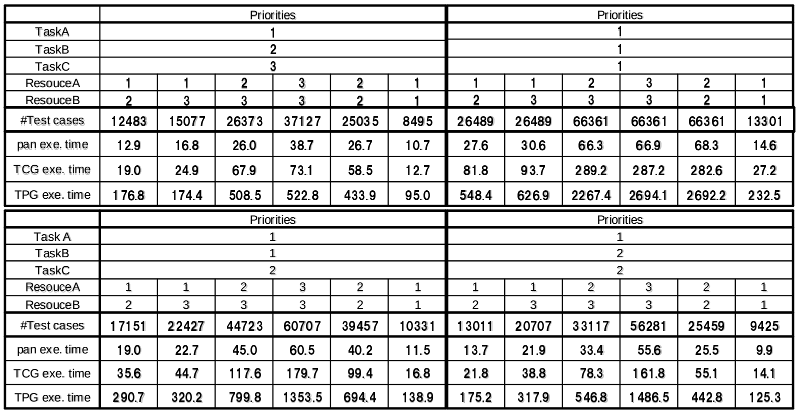
An important technical achievement of this study is that we succeeded in seamlessly connecting two verification activities; verification of a design model with model checking and testing of a product. A point to achieve this is to regard a design model as a test oracle after making much effort to verify it. We rely on the design model when we generate test cases and programs. In this approach, construction and verification of the design model are directly associated with testing of products which are recognized as an important activity in industries. That fact makes it easier to motivate engineers to construct a formal design model and use model checking to verify it. We could concentrate on constructing and verifying the design model by showing a way to effectively use them in developments.

It is ideal to apply formal methods to every phase of developments theoretically; however, it is often not feasible in practice due to their high cost. Therefore, it is important to find an activity to be concentrated on and apply formal methods in the activity. In addition, making the best use of an artifact obtained by the application of formal methods in other activities is also important. Such concentration and effective use of the artifact make the cost to apply formal methods reasonable in practice. It is needless to say that this is a trade-off between theory and practice. The degree of correctness will become less, in a sense, by restricting activities to apply formal methods. In our approach, we concentrate on the construction and verification of a design model in addition to adopting testing to check a product. As a result, we succeeded in making the cost reasonable so that model checking can be applied to a commercial product. The reason why we concentrated on the design model is that it is easy to characterize behavior of OSs like OSEK OS in an imperative specification language which is used to the design model. Actually, a ready queue appeared in the standard specification of OSEK/VDX to explain the behavior of the OSs. Such precise behavior of the OS is taken into account from an early stage of developments. Thus, constructing a formal design model is relatively natural in development.

B. Verification Results

Our project consists of two researchers of JAIST and several engineers of REL and DENSO. The construction and verification of the design model were conducted by the JAIST side in which both of two researchers are involved. The review of the design model and verification results was conducted by all of members of this project. We took around 6 months to construct the design model. Although we did not measure the exact period to obtain the design model, it must be actually much less than 6 months since they had not only this project but also the other works. On the other hand, REL is developing this kind of OSs including REL OS more than ten years. In addition, REL OS was plentifully verified by REL and DENSO before it was assembled in the current series of cars. Actually, they discovered and had fixed bugs many times at that point. REL OS was already sufficiently sophisticated when we started our project.

It was still surprising that no bug of REL OS was discovered in our verification because testing was conducted by a huge number of the test cases. Furthermore, we were surprised at the fact that the quality of the design model is as same as REL OS even though the development period of the design is much less than that of REL OS. Remind that the design model was constructed by the researchers of the JAIST side who have much less experience than the engineers of REL and DENSO



	Priorities						Priorities					
TaskA	1						1					
TaskB	2						1					
TaskC	3						1					
ResouceA	1	1	2	3	2	1	1	1	2	3	2	1
ResouceB	2	3	3	3	2	1	2	3	3	3	2	1
#Test cases	12483	15077	26373	37127	25035	8495	26489	26489	66361	66361	66361	13301
pan exe. time	12.9	16.8	26.0	38.7	26.7	10.7	27.6	30.6	66.3	66.9	68.3	14.6
TCG exe. time	19.0	24.9	67.9	73.1	58.5	12.7	81.8	93.7	289.2	287.2	282.6	27.2
TPG exe. time	176.8	174.4	508.5	522.8	433.9	95.0	548.4	626.9	2267.4	2694.1	2692.2	232.5

	Priorities						Priorities					
Task A	1						1					
TaskB	1						2					
TaskC	2						2					
ResouceA	1	1	2	3	2	1	1	1	2	3	2	1
ResouceB	2	3	3	3	2	1	2	3	3	3	2	1
#Test cases	17151	22427	44723	60707	39457	10331	13011	20707	33117	56281	25459	9425
pan exe. time	19.0	22.7	45.0	60.5	40.2	11.5	13.7	21.9	33.4	55.6	25.5	9.9
TCG exe. time	35.6	44.7	117.6	179.7	99.4	16.8	21.8	38.8	78.3	161.8	55.1	14.1
TPG exe. time	290.7	320.2	799.8	1353.5	694.4	138.9	175.2	317.9	546.8	1486.5	442.8	125.3

Fig. 8. Test Results

with respect to OS developments. We expected before starting the testing that if testing would fail, that should be due to bugs of the design model or misunderstanding of behavior of REL OS. However, all the test cases have been passed at a time. There was no backtrack to improve the design model once we started the testing. We succeeded in making the design model whose quality is similar to REL OS in six months. We can say from this fact that model checking effectively works for ensuring the quality of the design model. Although an objective of the testing is not to check the design model but the implementation, effectiveness to apply model checking to the design model has been proved consequently.

Our approach is based on exhaustive search methods. We encounter state explosion problem as far as we use those methods. Thus, we introduced techniques to prevent the state explosion problem as follows. Firstly, we bounded variations of configurations such as the numbers of tasks and resources in the verification of the design model. Secondly, we separated the possible behavior of environments into twelve cases. Finally, we bounded variations of configurations in the testing. We could not guarantee to cover the whole behavior of the design model and implementation due to the techniques. In addition, the verification of the design model relies on the environment models. Even though the environment models are simpler than the design model, they might still be wrong. Our verification depends on the validity of such bounds and environment models.

To convince that they are valid, we made effort to review the environment models and variations of configurations. There are techniques to make the validity more convincing. For example, we use theorem proving to verify the design model for unbounded variations. We construct a formal specification of OSEK OS, then verify the design model and environment model against it to make sure that they meet the specification. On the other hand, we have to pay additional cost if we adopt those techniques furthermore. We need to carefully decide what techniques we should take from not only the theoretical but also practical point of view. The combination of techniques including model checking and testing as shown in this paper is an approach whose cost is acceptable in the field of automotive systems.

C. Meaning of Testing

As mentioned so far, REL OS was already plentifully verified REL and DENSO. This verification was conducted based on ordinary testing methods. Some bugs were found with respect to the scheduling of REL OS then. We confirmed that those bugs could be also discovered by testing with our approach. As all the test cases have been passed, we can say that no such bugs exist within behavior represented by them.

Test cases generated by our approach contain invocation sequences of API functions which we do not usually make because they are obtained by searching reachable states of non-deterministic invocation of those functions. Such test cases allow us to check behavior which is not realized in current applications but will be realized in future ones. This is similar to acceleration testing applied in the field of materials. One can say that we conducted acceleration testing of software in a sense. We think that this testing is important for OSs since they are used in various ways for a long time.

A test case generated can be regarded as an application performed with REL OS. In this sense, we gained evidence to perform a number of applications on the OS. The evidence is important for satisfying safety standards such as IEC61508[5] and ISO26262[4].

D. Creating the confidence in correctness

In the verification of seL4 kernel [20], [21], C implementations of the kernel as well as specifications described in Haskell were automatically translated into descriptions of Isabelle/HOL according to pre-defined translation rules. The kernel was verified based not on the implementations themselves but on the translated descriptions. Theorem proving with Isabelle/HOL allows us to create the strong confidence in the correctness of proofs done in the verification. On the other hand, a gap between the descriptions of Isablelle/HOL and the implementations still remains. In our approach, the implementation of REL OS was exhaustively executed within specific bounds in testing. We think that executing the implementation itself is very important in order to create the confidence in its correctness. Even though the translated descriptions are verified, it is unimaginable to release the implementations without executing them. Although it is impossible to check all of cases which may happen in the testing, the testing provides evidence that the implementation really works well.

In the existing works on the verification of OSEK OS [23], [24], [27], [28], source codes are only targets to be verified. Since we suspect that the source codes might be wrong, we need another description which we rely on. In our approach, we rely on the design model which was verified by the model checking. We made much effort to verify the design model so that we could agree that it realized correct behavior of the OS. That is, we created the confidence in the correctness via such design model.

VIII. Conclusion

Applying formal methods to developments of commercial products is often recognized as hard in industries. In fact, we have the experience to educate engineers in formal methods[31], [32] and heard such opinions from many of them. In order to persuade them so that formal methods can be practically applied, it is important to show a successful case study of a practical system. In this paper, we showed a case study that model checking, which is one of formal methods, was applied to an automotive OS. What we should emphasize here is that our target is a commercial product, that is, REL OS. In addition, engineers who develop the product are involved in this project. Showing evidence that model checking has been successfully applied to the commercial product is a primary contribution of this paper.

We encountered various pragmatical problems in the application of model checking as shown in the paper. We solved them by combining with engineering techniques such as review and testing. We used informal methods with formal methods for making our approach practical. No bug was found as a result; however, we obtained the confidence that the quality of REL OS is very high. We think that obtaining the confidence is quite different from finding bugs by testing. Clearly, the former is much harder than the latter. The exhaustive techniques that we have adopted allow us to convince that REL OS is correct.

The OS is going to use in a next series of cars not only for parts which it is currently embedded to but also the other ones. We are convinced that REL OS performs correctly even for the other parts since we have conducted the exhaustive testing which can be regarded as the acceleration testing. The same approach is being applied to another function of REL OS. We continue to verify it and extend the approach so that we can acquire more confidence with respect to the quality.

References

[1] Technical Assessment of Toyota Electronic Throttle Control Systems, NHTSA, 2011.

[2] OSEK/VDX Operating System Specication 2.2.3., 2005.

[3] Specification of Operating System 4.0.0, AUTOSAR, 2009.

[4] ISO 26262 Road vehicles - functional safety, 2011.

[5] IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems, 1998.

[6] L.Sha, et.al: Priority Inheritance Protocols: An Approach to Real-Time Synchronization, IEEE Transactions on Computers 39 (9), pp.1175–1185, 1990.

[7] G.J.Holzmann: The Spin Model Checker, 2004.

[8] J.Penix, et.al: Verifying Time Partitioning in the DEOS Scheduling Kernel, Formal Methods in System Design, Vol.26, No.2, pp.103–135, 2005.

[9] C.Pasareanu: DEOS Kernel: Environment Modeling using LTL Assumptions, Nasa ames technical report nasa-arc-ic-2000-196, NASA Ames Research Center, 2000.

[10] M. Dwyer and C. Pasareanu: Filter-based model checking of partial systems, Foundations of Software Engineering, pp.189–202, 1998.

[11] Object: Unified Modeling Language: Superstructure, version 2.1.2, 2007.

[12] J. Warmer and A. Kleppe. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1999.

[13] K.Yatake and T.Aoki: Automatic Generation of Model Checking Scripts based on Environment Modeling, International SPIN Workshop on Model Checking of Software, pp.58–75, 2010.

[14] K.Yatake and T.Aoki: SMT-based Enumeration of Object Graphs from UML class diagrams, International workshop UML and Formal Methods, ACM SIGSOFT Software Engineering Notes, 37(4), pp.1–8, 2012.

[15] K.Yatake and T.Aoki: Model Checking of OSEK/VDX OS Design Model Based on Environment Modeling, International Colloquium on Theoretical Aspect of Computing, pp.183–197, 2012.

[16] D.Lee and M.Yannakakis: Principles and Methods of Testing Finite State Machines - a Survey, Proceedings of the IEEE, vol. 84, no. 8, pp.1090–1123, 1996.

[17] J.Chen and T.Aoki: Conformance Testing for OSEK/VDX Operating System Using Model Checking, Asia-Pacific Software Engineering Conference, pp.274–281, 2011.

[18] G.Fraser, F.Wotawa and P.Ammann: Testing with Model Checkers: A Surv ey. Journal for Software Testing, Verification and Reliability, Volume 19 Issue 3, pp.215–261, 2009.

[19] G.Klein: Operating System Verification - An Overview, Sādhanā , 34(1), pp.26–69, 2009.

[20] G.Klein, et.al: seL4: Formal verification of an OS kernel, ACM Symposium on Operating Systems Principles, pp.207–220, 2009.

[21] G.Klein, et.al: Comprehensive formal verification of an OS microkernel ACM Transactions on Computer Systems, Volume 32 Issue 1, pp.1–70, 2014.

[22] J.Penix, et.al: Verifying Time Partitioning in the DEOS Scheduling Kernel, Formal Methods in System Design, Vol.26, No.2, pp.103–135, 2005.

[23] L.Zhu, et.al:Formalizing Application Programming Interfaces of the OSEK/VDX Operating System Specification, Theoretical Aspects of Software Engineering, pp.27–34, 2011.

[24] Y.Huang, et.al:Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP, Theoretical Aspects of Software Engineering , pp.142–149, 2011.

[25] E.Cohen, et.al: VCC: A Practical System for Verifying Concurrent C, International Conference on Theorem Proving in Higher Order Logics, pp.23–42, 2011.

[26] PAT, Process Analysis Toolkit 2.9 User Manual. Software Engineering Lab, School of Computing, National University of Singapore, 2007.

[27] Y.Choi: Safety Analysis of Trampoline OS Using Model Checking: An Experience Report, International Symposium on Software Reliability Engineering, pp.200–209, 2011.

[28] Y.Choi: Model checking Trampoline OS: a case study on safety analysis for automotive software, Software Testing, Verification and Reliability, vol.24, Issue 1, pp.38–60, 2014.

[29] Trampoline - open source RTOS project, http://trampoline.rtssoftware.org.

[30] J.M.Spivey:The Z notation: a reference manual, 1992.

[31] Y.Tahara, N.Yoshioka, K.Taguchi, T.Aoki and S.Honiden: Evolution of a course on model checking for practical applications, ACM SIGCSE Bulletin, Vol.41 , Issue 2, pp.38–44, 2009.

[32] H.Nishihara, K.Shinozaki, K.Hayamizu, T.Aoki, K.Taguchi and F.Kumeno:Model checking education for software engineers in Japan, ACM SIGCSE Bulletin, Vol. 41 , Issue 2, pp.45–50, 2009.