Formal Model-Based Test for AUTOSAR multicore RTOS

Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do and Hitoshi Ohsaki
National Institute of Advanced Industrial Science and Technology
Email: {fang-ling, t.kitamura, do.ngoc}@aist.go.jp, ohsaki@ni.aist.go.jp

Abstract

AUTOSAR multicore RTOS is a safety-critical concurrent system, for which high quality is required. A conformance test is important to ensure the quality of the software, but the conventional test is low in coverage and high in cost. In this paper, we present a formal model-based test for multicore RTOS that supports AUTOSAR specifications. First, we developed a formal model. With the model, we developed a test case generator, from which an entire test suite can be extracted. Moreover, we proposed a test program generator, with which optimal executable test programs can be generated fully automatically. Both of the generators are assisted with model checking on the formal model. Bug analysis also becomes easy.

Our method demonstrated its advantage over conventional testing by finding 33 test cases for three system service calls, whereas a conventional test carried out by a development team found only 10 test cases. Our method can improve the coverage of the test, clearly saving in cost and development time. It is expected to significantly improve the testing of the AUTOSAR multicore RTOS.

Keywords

AUTOSAR multicore RTOS, model checking, SPIN, model-based test, test automation.

1. Introduction

AUTOSAR (AUTomotive Open System ARchitecture) [1] aims for an industry standard for distributed control units in vehicles. RTOS (Real-Time Operating System) [2] is an operating system intended to serve real-time application requests. The AUTOSAR standard RTOS is safety-critical, but its correctness is severely compromised particularly when it runs concurrent processes on multicore.

Currently, conventional tests designed by hand are still the main method for detecting the bugs, and these tests have several problems; for example, ensuring a perfect coverage of the test is difficult, particularly for the severe timing between different cores. Also, the manual design for the execution of test cases is an inherently time-cost process, and it must be carried out for every system with a different configuration. Moreover, bug analysis usually also consumes much time.

In this paper, we propose a formal model-based test method to address the conventional testing problems of AUTOSAR multicore RTOS. Our method aims at a more complete test suite as well as automation of the test, which can significantly improve quality, cut cost, and shorten the time of the test.

Figure 1 shows the system process outline. The method mainly involves a formal model describing the AUTOSAR RTOS requirements. With a configuration for the run-time environment of each ECU (Electronic Control Unit) [2], a concrete model for a system is created. Using the test case generator, an exhaustive test suite which is a set of test cases with the expected outcome (test oracle) is achieved from the model. Consequently, an entire (it means the completeness allowed by the limited computer resource) and feasible (it means realistic processing time) test suite can be obtained after filtering with the coverage criteria. Finally, with the test program generator, the optimal test sequence for every test case is calculated, and then translated to execution program. If a bug exists, analysis is easy because it can be carried out step-by-step along the execution path provided from the model checker.

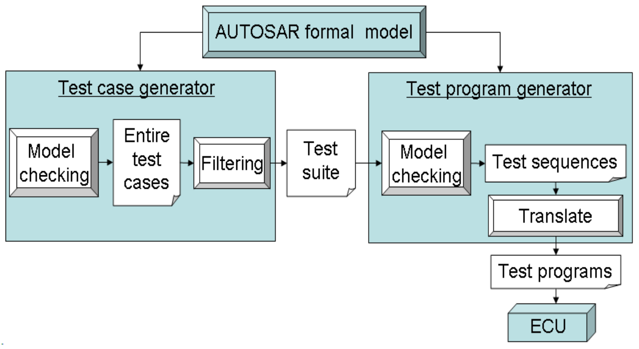

Fig. 1. System outline

The remainder of this paper is structured as follows: Section 2 reviews several related works. Section 3 explains the preliminaries regarding linear temporal logic (LTL) and AUTOSAR RTOS. Section 4 introduces how to construct a formal model that conforms to the AUTOSAR RTOS. Section 5 demonstrates how to generate a test suite from the model with the test case generator, and how to achieve executable test programs with the test program generator. The debug method is also introduced. Sections 6 provides the results of our experiment. In Section 7, the paper concludes with a brief discussion about our contribution and future research.

2. Related work

The use of model checking for testing has been investigated for at least a decade [3, 4] and a significant amount of research has been written on this subject. Most researchers proposed a test-case generation method by verifying a temporal logic formula specifying reachability to a particular state [5, 6, 7, 8]. For example, [6] iteratively extended already discovered tests and thus extended additional goals. [7] enumerates the state wherein the user must provide a CTL formula [9] for every state to be covered. There are also many existing tools for the model-based test [10, 11, 12]. Our method differs from these previous works and with the original model in that the particular states concerned with the test cases can be extracted more easily, and state space is suppressed.

The MODISTARC conformance test project [13] includes OSEK specifications for the conformance test. The test suites are specified by a classification tree, which is a test method for the systematic design of test cases based on their specification. Our method is different because it is not based on the constraint conditions of the test, but on the behavior of the system.

Toppers project[1] reported the test environment of multicore RTOS. The fundamental difference between our study and their study was that their method used a formal specification for a test scenario, which needed analysis manually on about 59 thousand lines, whereas our method is conducted on a formal model specified for RTOS requirements. Our method does not need to consider a complex test scenario.

Many other works exist that try to verify the implementation of an RTOS. For example, OpenComRTOS[2] develops a system concurrently with a formal model to certify that implementation meets its specification. [14] is a formal refinement proof linking an abstract specification to its C implementation. Research [15] develops a real-time operating system FreeRTOS with the B method [16]. However, verifying the system is costly and complex.

There has not been a formal model-based test, which includes both automatic test case generation, and a test-case execution, for AUTOSAR multicore RTOS until ours.

3. Preliminaries

3.1. Linear Temporal logic (LTL)

Kripke structure is used to interpret the semantics of LTL [9] and to describe model checking.

Kripke Structure: A Kripke structure is a 4-tuple $M, \pi = \langle S_0, S, R, L \rangle$, where S is a set of states with the initial state S_0, R is a transition between states, and L is a mapping from the states to atomic propositions. $L(s)$ is a set of atomic propositions that holds for a state $s \in S$. A path $\pi : \langle s_0, s_1, ... \rangle$ of Kripke M is an infinite or finite sequence such that $\forall i \geq 0 : (s_i, s_{i+1} \in R)$ for M. The LTL formula ϕ that holds in the path π of M is represented as $M, \pi \models \phi$.

LTL Syntax: The BNF definition of the LTL formulas is given below:

$$\phi, \psi ::= true \mid false \mid P \mid \\ \phi \wedge \psi \mid \phi \vee \psi \mid \neg\phi \mid \circ\phi \mid \Diamond\phi \mid \Box\phi$$

P is the atomic proposition, $\circ$ represents $next$, $\Diamond$ represents $future$, and $\Box$ represents $global$.

LTL Semantics: Satisfaction of LTL formulas by a path π is inductively defined as follows.

$\pi \models P$ iff $P \in L(\pi_0)$
$\pi \models \neg\phi$ iff not $\pi \models \phi$
$\pi \models \phi \wedge \psi$ iff $\pi \models \phi$ and $\pi \models \psi$
$\pi \models \phi \vee \psi$ iff $\pi \models \phi$ or $\pi \models \psi$
$\pi \models \circ\phi$ iff $\pi_1 \models \phi$
$\pi \models \Diamond\phi$ iff $\pi_i \models \phi$ for some $i \geq 0$
$\pi \models \Box\phi$ iff $\pi_i \models \phi$ for any $i \geq 0$

SPIN [17] is a model checker which is the most mature open-source software tool available for temporal logic LTL. Promela [17] is the specification language of SPIN. Promela allows specification of distributed automatons (called transition systems in this paper) which can communicate using either message channels or shared memory. Promela is suitable to describe the concurrent actions of multicore RTOS.

3.2. AUTOSAR RTOS

For construction of the RTOS model, the first stage is to analyze the object system requirements from the prospect of purpose. As our test purpose is to examine the behavior of OS service operation, it is reasonable to group the requirements according to service functions as: task management, interrupt processing, event mechanism, resource management, etc.

3.2.1. General concepts of RTOS. This section discusses several general AUTOSAR RTOS concepts. Further details can be referred to in the document [1, 2].

Tasks and system service calls (SVCs): A task is a basic unit of programming controlled by an RTOS and refers to an execution context of RTOS management or an application. The RTOS provides facilities such as priorities and service calls to control the execution of tasks. Each task has a priority, a preemptive policy, and a statically assigned set of events, alarms, and resources. Figure 2 shows that the task transits between four states: $running, ready, waiting$, and $suspended$[3].

SVCs are Application Program Interfaces (APIs) used to request service from the operating system. For example, each task may activate other tasks using the $activateTask$. $TerminationTask$ transfers the currently $running$ task into the $suspended$ state and triggers the scheduler to execute the next $ready$ task, or an internal idle loop, if there is no task. $ChainTask$ activates other task and then terminates itself ($chainTask$ does not appear in the Figure 2 because it can be separated as $activateTask$ and $terminateTask$). $WaitEvent$ causes the $running$ task to enter the $waiting$ state. A task in $waiting$ state transits to $ready$ while it receives an expected $setEvent$ and then clears it with $clearEvent$.

1. Toppers project: http://www.toppers.jp/en/index.html.
2. OpenComRTOS: http://www.altreonic.com.
3. $activateTask, terminateTask, chainTask$ are referred as $activate, terminate, chain$; $running, suspended, waiting$ are referred as $run, sus, wait$; task is referred as T in the following examples.

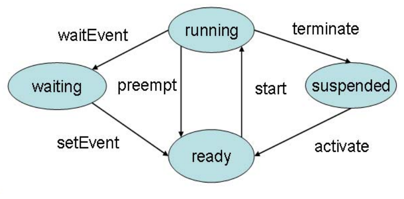

Fig. 2. Task transition

Preemptive policy and scheduler: The Central Processing Unit (CPU) can be occupied by only one task at a time. Preemptive policy means when the running task's priority is lower, it will be replaced by a $ready$ task with a higher priority.

The scheduling policy manages the task execution according to the state of tasks, preemptive policy, SVCs, etc. This enables minimal interrupt latency and minimal switching latency, to respond more quickly or predictably thereby ensuring task performance in a given period of time.

Interrupt service routine (ISR): Two categories of ISR are defined. ISR category 1 (ISR1) has no effect on the operating system. ISR category 2 (ISR2) calls API functions and effects on the operating system because rescheduling is initiated by the generated ISR2 epilogue.

Event: Events provide a method for the synchronization of different tasks, or ISR2. Events are statically allocated and assigned to an extended task. When a task is activated, all its events are cleared and the task transits to the $waiting$ state by $waitEvent$, and escapes from $waiting$ state when the expected $setEvent$ is on.

Resource: A resource is used to ensure the shared access. Priority Ceiling Protocol (PCP) [1] is adapted for resources to avoid priority inversion and deadlock. $getResource$ is used to acquire a resource, and $releaseResource$ is used to release it.

Lock and critical section: An important part of an RTOS is the lock mechanisms, and the Critical Section (CS) implementation [2]. Critical section means the way of arranging things so that a modification of shared data structure appears atomic to ensure consistent processing. Lock is a binary semaphore used to implement critical section.

3.2.2. Configuration and typical task behavior. The AUTOSAR RTOS uses a layered software architecture that includes the basic software layer, runtime environment and application layer to make a component-based design possible. The run time environment parameters are configured for each ECU (Electronic control unit) [2]. For example, the number and identity of tasks, the task's priority, assigned resources and events, etc. Table 1 shows a simple example of a configuration which is relevant to our experiment (there are also many other configuration items such as basic or extend task for event, resources priority, etc.)[4]. Figure 3 shows the typical behavior of the RTOS.

TABLE 1. System configuration

	Core0			Core1					
	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
pri	3	2	1	2	1	1	0	0	0
pre	F	F	F	F	F	F	N	F	N
ini	run	sus	sus	run	sus	sus	sus	sus	sus

prio:priority; pre:preemptive policy; F:full; N:non; ini:initial status

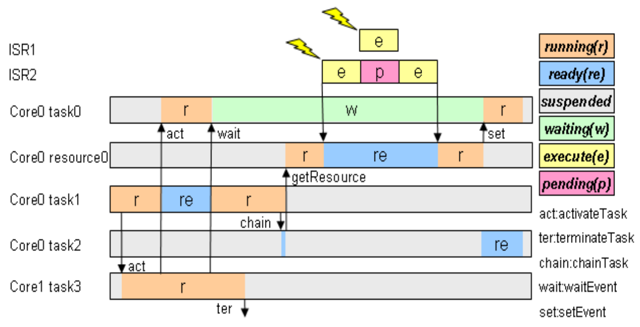

Fig. 3. Typical behavior of the RTOS

4. Formal model of AUTOSAR RTOS

The model creation, however, is one of the most difficult parts of the whole development process. Model abstraction is essential, both for performance and for scalability reasons. As the test purpose is to examine SVCs in a special system state of the system, we abstract the AUTOSAR RTOS from the perspective of system transitions driven by SVCs.

In this section, we illustrate how to construct the formal model of the system described in Section 3.2, before introducing how to use it for a test in Section 5.

Corresponding the component-based designed AUTOSAR RTOS software architecture, we construct an abstract model of the basic software layer, and then create a concrete model with the runtime environment configuration. Verification for the formal model is also demonstrated at the last of this section.

4.1. Definition of the model

Here we must construct a finite model $M, \pi = \langle S_0, S, R, L \rangle$. Corresponding to the AUTOSAR RTOS software architecture, we first define an abstract model $M, \pi = \langle S, R, L \rangle$, which describes the requirement of AUTOSAR RTOS. The concrete target model corresponding to the run time environment is created with the configuration of the system as the initial state S_0.

4. All the examples throughout the paper assume a system with the configuration of Table 1. For comparison, the configuration is similar to what is used by the RTOS development group.

With the specification language Promela, the whole system can be decomposed as several transition systems (distributed automatons) which communicate with each other. Each transition system is an automaton with states, and labels to indicate the condition of transitions. Definition is performed in two steps as follows.

- Define states and transition condition labels for each transition system.
- Define the interaction communication between the transition systems.

4.1.1. The transition systems. We defined task, ISR2, CPU, resource, and lock as transition systems.
Task transition system: Task transits between $running, ready, suspended, waiting,$ and $ceiling$ with the labels sharing the same name in documents [1] for transitions caused by SVC commands, as shown in Figure 4. Different from the transitions of the task in Figure 2, the

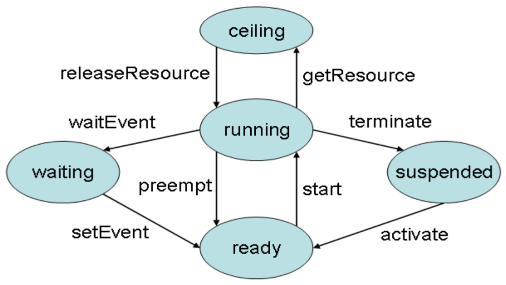

Fig. 4. Task transition system

state $ceiling$ distinguishes an abnormal $running$ state when occupying a resource. The labels $preempt, start$ are used for implicit task transitions between $running$ and $ready$ because of the preemptive priority policy. We define the labels corresponding to the SVCs used in the task transition system as a set:
$SVCs : \{activate, terminate, chain, waitEvent,$
$setEvent, clearEvent, getResource, releaseResource\}$.
ISR transition system: The ISR2 state transits between the $executing$ and $pending$ states, depending on whether the ISR2 occurs. The ISR2 priority is higher than all tasks, and it will be processed immediately when it occurs, as shown in Figure 5.
CPU transition system: The CPU state transits to the state $occupied$ when it accepts get, and to the state $free$ when it accepts put from a task which is occupying the CPU. The CPU is provided with a ready queue, which holds task identifiers in $ready$ state, and the task identifier is allocated in order of priority. The CPU executes the task at the beginning of the queue, when the CPU is $idle$. If the priority of the task $running$ on the CPU is lower than the first task in the ready queue, and it is preemptive, then transitions occur, the $running$ task transits to $ready$ and the $ready$ task transits to $running$, as shown in Figure 6.

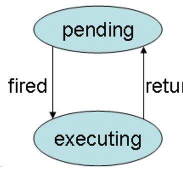

Fig. 5. ISR transition system

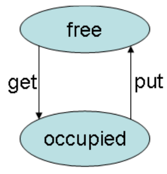

Fig. 6. CPU transition system

Resource transition system: Resource transits between $occupied$ and $free$ states depending on whether it is occupied with the labels get or released with put by a task.

According to Priority Ceiling Protocol (PCP) [1], the task's priority temporarily changes when occupying the resource to prevent priority violation. Moreover, a task can require several resources, but a resource cannot be required by more than one task.
Lock transition system and critical section: Because RTOS application function's operation is deterministic and the interpretation can be directly applied, all the SVC transitions are processed atomically in the model. This means a critical section for SVC transitions without a branch until the transition finishes.

As mentioned in Section 3.2, lock is a binary semaphore [17] which can be used to ensure a critical section. We define the critical section in the model as:
$CS : \langle s_i, s_j \rangle \in \pi \wedge \langle s_i, s_{i+1} \rangle \in SVCs \wedge \langle s_l, s_{l+1} \rangle \notin SVCs (i < l < j)$. S_i is the start point of critical section with a transition of SVCs, following with several transitions $\langle s_l, s_{l+1} \rangle$ before ending at s_j.

Figure 7 is an example for the explanation of the critical section. The path between two circles is a critical section, representing a complete process for a SVC transition. The transitions leaving a circle belong to the set $SVCs$, the other transitions including $put, get, start, preemt, \ldots$ assist to complete the SVC process. The definition of the critical section substantially differs in our method from other previous works because it greatly suppresses the state space.

The explosion problem means the size of the state space grows exponentially with the number of processes. It is extremely important when adopting the model checking technique. Usually the state space is $O(S^n)$, S is the size of each transition system, and n is the number of the transition systems. Because the states inside a critical section are treated as an unbranched process, the transitions $put, get, start, preemt, \ldots$ assisted with the transition systems do not contribute to the space explosion anymore. In another word, n in $O(S^n)$ is reduced to the number of task transition systems.

Moreover, the exhaustive test case set can be extracted from the ends of the critical section (circles in Figure 7) simply without model checking for LTL formulas.

4.1.2. Communication between transition systems. The transition systems interact with each other through communi-

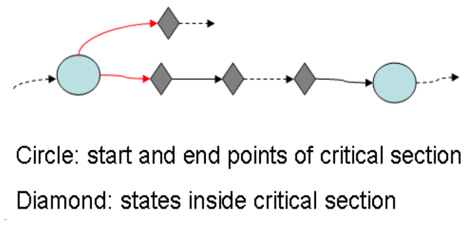
Circle: start and end points of critical section
Diamond: states inside critical section

Fig. 7. Critical section

cation. When the transition condition message is passed into the transition systems, the message causes or constrains the transitions of the systems. Channels are used to send and receive messages. There are asynchronous and synchronous communications according to the requirements of communication.

Asynchronous communication: Asynchronous communication means the sender can send the data at any time, without concern about when the data is received. A queue is required to store the data until it is received.

Asynchronous communication is used between tasks or from ISR2 to a task. For example, when the transition condition message $activate$ is sent from $task_3$ to the $task_1$, and $task_1$ is in $suspended$ state, $task_1$ responds and then transits to the $ready$ state. However, $task_3$ does not concern when the transition of $task_1$ occurs after it sends the message.

Synchronous communication: Synchronous communication is direct communication, where all parties involved in communication are present at the same time. In each of these communication processes, the sender must wait until the receiver responds. A queue is not necessary if the communication is synchronous.

Synchronous communication is used between tasks and the CPU, tasks and resources, and tasks and locks, in which the occupation required from the task for CPU, resource, and locks must be exclusive. The transition condition message get is used to occupy and the message put is used to release them. Moreover, the implicit preemptive task transition conditions $preemp$ and $start$ are also used for synchronous communication by the CPU.

For example, $task_0$ is $ready$ but its priority is higher than running $task_1$, $task_0$ sends a message get to the CPU. CPU sends $preempt$ to $task_1$, and then accepts a put message from $task_1$. The CPU then accepts the get message from the $task_0$ before sending the message $start$ to $task_0$. When all the synchronous communications have finished, $task_0$ transits to $running$ and $task_1$ transits to $ready$.

4.2. Create a concrete model

While in Section 4.1 an abstract model conforming the requirement of AUTOSAR RTOS is described, the concrete model must be created with the system configuration. In this manner, our abstract model can be reused for any RTOS system conforming AUTOSAR requirement.

The configuration contains the information of number, identities, priority, accessibility of tasks, resources, ISR2, and events, and etc. The following is an example that shows a part of configuration file used in our experiment.

$T_0.priority = 3$ /$task_0$'s priority is 3/
$T_0.coreID = 0$ /$task_0$ is a task on $core_0$ /
$T_0.preemptive = preemptive$ /$task_0$ is preemptive/
$T_0.state = suspended$ /$task_0$'s initial status is suspended/
...

The RTOS model usually includes several CPU, tasks, resources and several states for every transition system, that means the model is finite and feasible.

4.3. Verify the model

As the model is defined with Promela, model checking can be carried out to detect deadlock, livelock and assertion, and it is also used to verify properties that can be described with the LTL formulas. The process named CEGAR (Counterexample-guided Abstraction Refinement) [9] is iterated until violations in Figure 8 are not reported.

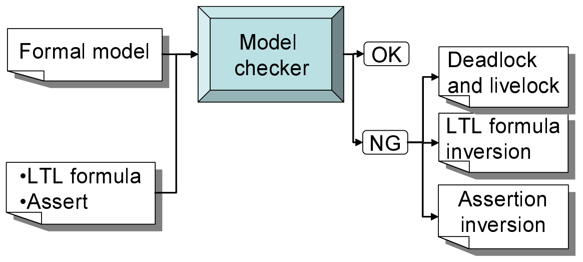

Fig. 8. Verification for the model

Deadlock and livelock: Deadlock and livelock can be detected with a model checker automatically. Deadlock is likely to occur in the multicore RTOS, where it is a fatal system error and not tolerated by the automotive system.

LTL formula: LTL formulas that specify linear properties such as reachability, safety, or other properties of the RTOS and SVCs can be verified. The following LTL formula is an example expressing priority inversions that RTOS must not violate, i, j are free variables that must be bound to the identities of task ($task_i.priority > task_j.priority$, $task_i$ and $task_j$ belong to the same core).

$$\Box\neg(\ task_i = ready \wedge task_j = running) \tag{1}$$

Assertion: Assertions are conditions inserted at trap points of the model, whenever the model checker traverses the point it will check if the conditions are violated. The following are two examples used in our work (n is the number of tasks on the CPU):

$$\neg(\sum_{i=0}^{n} task_i.state = ready > 0 \wedge CPU = free) \quad (2)$$

$$\neg(\sum_{i=0}^{n} task_i.state = running > 1) \quad (3)$$

Formula (2) means the state must not exist that a CPU is free but there is ready task. Formula (3) means only one task can be running on one CPU.

Both the model checking for LTL formulas and the trap points for assertions must be designed carefully only for the ends of critical sections (circles in Figure 7), because erroneous report of violation such as a priority violation is possible when the transition process has not been completed.

5. Use the formal model for tests

Model checkers are formal verification tools, capable of providing counterexamples to violated properties. If the system model has finite states, the exploration may be conducted automatically using a model-checking algorithm [9]. Our idea of a test with a model checker is not only to interpret counterexamples to find test cases, but also to interpret the test cases as counterexamples to find the execution sequence.

As mentioned previously in this paper, the test purpose of our work is to examine whether the SVCs can result in correct transitions of the system.

Test Case: A test case of model m contains pre-state(s_{pre}) and post-state(s_{post}) for a SVC operation(op) which has an executor task(E) and the affected task(A). It is defined as:
$t(m) = \langle E, op, A : s_{pre} \rightarrow s_{post} \rangle$,
$\pi : \langle s_{pre}, \ldots, s_{post} \rangle \in CS, op \in SVCs$.
Test Suite: A test suite is a finite set of n test cases $TS(m) = \bigcup_{i=0}^{n} t_i$.
Test Sequence: A test sequence $Seq(m, t_n) : \langle t_0, t_1, ...t_n \rangle$ is a finite path involving several consequent test cases such that $\forall i : (0 \leq i < n) : s_{post}(t_i) = s_{pre}(t_{i+1})$.

In this section, we explain how to carry out a completely automatic test with model checking (model checker) as shown in the following stages.

- With the test case generator
  - Extract the heuristic test cases set from the formal model.
  - Filter the feasible and entire test suite with coverage criteria.
- With the test program generator
  - Calculate the test sequences with model checking for the test cases.
  - Translate the test sequences into the executable test programs.

The test programs include the correct output for every step of SVC's execution. The actual outputs are compared with expected outputs, and bug analysis is easy with the execution trace.

5.1. Extract a set of test cases

A heuristic and finite test case set can be extracted from the formal model, when the model checker effectively explores the whole state space of the system. As mentioned before, the test case is the pre-state and post-state of a transition for SVC, which means the system state where the SVC starts and ends must be extracted. We set $task_0 = claim$ ($claim$ is any term which is not a normal state of the task) as a trap property, set the trap point at the end of critical section, and set $\square\neg(task_0 = claim)$ as a counterexample. Therefore only and all of the trap points will violate the property and catch the traversal when exhaustive model checking is carried out. The following is an example of a test case $t(m) = \langle E, op, A : s_{pre} \rightarrow s_{post} \rangle$ (E is T_0, A is T_1, op is $activate$, s_{pre} is $T_0.run, T_1.sus, T_2.sus$, s_{post} is $T_0.run, T_1.ready, T_2.sus$).

$T_0\ activate\ T_1$:
$T_0.run, T_1.sus, T_2.sus \rightarrow T_0.run, T_1.ready, T_2.sus$

This test case represents the system state changes from $[task_0.run, task_1.sus, task_2.sus]$ to $[task_0.run, task_1.ready, task_2.sus]$ after $task_0$ activates $task_1$.

Positive testing is testing which attempts to show that a system does what it is supposed to do. Because the model conforms to the requirements of AUTOSAR RTOS, only the valid input for positive testing can be accepted[5].

Usually, huge amounts of test cases can be found that contain a significant redundancy, and scalability is an issue in practice. Next, we explain how to filter the heuristic test cases into a entire and feasible test suite.

5.2. Test selection strategies

Test-selection criteria is essential for a practical and entire test suite. Our method involves classification of the heuristic test case into subsets by considering various condition's combinations of input, output and SVCs' operations, and then selects one case from every subset, which presents one kind of combination.

In our current work, we used a classification tree in a similar manner as the OZEK test plan [13]. The classification tree contains conditions $C_1, C_2, \ldots$ as follows.
C_1 : executor and affected tasks are on the same core.
C_2 : executor task is preemptive.
C_3 : executor task's operation is activate.
C_4 : executor task's priority is higher than affected task's.
...
The classification is used to find all the possible combinations of the conditions from the heuristic test case set. For example, the following two cases, (4) and (5), are classified as the task management test case 4 of the document [13], in which the

5. Negative testing is testing which attempts to show that the system does something that it is not supposed to do, but this negative testing is not within the contents of this paper.

high priority preemptive task activated a low priority task when the affected task's state is $suspended$.

$$T_0\ activate\ T_1 : T_0.run, T_1.sus \rightarrow T_0.run, T_1.ready \quad (4)$$
$$T_0\ activate\ T_2 : T_0.run, T_2.sus \rightarrow T_0.run, T_2.ready \quad (5)$$
$$T_0\ activate\ T_1 : T_0.run, T_1.sus, T_2.sus \rightarrow T_0.run, T_1.ready, T_2.sus \quad (6)$$
$$T_0\ activate\ T_1 : T_0.run, T_1.sus, T_2.ready \rightarrow T_0.run, T_1.ready, T_2.ready \quad (7)$$

The user can also define another criteria. For example, the criteria treats cases (4) and (5) as different cases because the affected tasks are different. However cases (6) and (7) belong to the same case 4 of the document [13] with this criteria, because T_2 is not relevant to the transition.

When all test cases in the heuristic set are classified, one case is selected from every subset as a member of our target test suite (TS). The size of TS is the number of possible combinations. The coverage of TS depends on the completeness of test cases achieved from the model checker. It is directly proportional to the maximum size of state space, which is the best effort of the exhaustive search. Although this coverage cannot be ensured by a complete state space, it is greatly improved over the manually designed test.

5.3. Test program generation

When applying a test case it is necessary to find a sequence of operations to set up the internal state before the test case can be applied. In other words, the test sequence is necessary because it guides the system to eventually execute the test case within the embedded environment.

Model checking can assist us here again, and provide support for finding appropriate paths (test sequences) [18]. In this stage, the test case t_i is used as counterexample, and model checking is carried out to find out the optimal path $Seq(t_i)$ to the case automatically[6]. For example, in our experiment when the case was as follows:

$$T_1\ chain\ T_0 : T_0.sus, T_1.run, \ldots \rightarrow T_0.run, T_1.ready$$

The optimal path found by the model checker was:

$$T_0\ activate\ T_1 : T_0.run, T_1.sus \rightarrow T_0.run, T_1.ready$$
$$T_0\ activate\ T_1 : T_0.run, T_1.ready \rightarrow T_0.run, T_1.ready$$
$$T_0\ terminate : T_0.run, T_1.ready \rightarrow T_0.sus, T_1.run$$
$$T_1\ chain\ T_0 : T_0.sus, T_1.run \rightarrow T_0.run, T_1.ready$$

But the shortest path with manual analysis for this case should be:

$$T_0\ activate\ T_1 : T_0.run, T_1.sus \rightarrow T_0.run, T_1.ready$$
$$T_0\ chain\ T_1 : T_0.run, T_1.ready \rightarrow T_0.sus, T_1.run$$
$$T_1\ chain\ T_0 : T_0.sus, T_1.run \rightarrow T_0.run, T_1.ready$$

6. Although the model checker SPIN declares it can find out the shortest path automatically, the path found with the function usually is not the shortest but short enough. We call it optimal path.

Usually there is the problem of mapping test sequences to the level of executable concrete test sequences. Because our model corresponds a concrete run time environment, the sequences from the the model are executable, which make it possible to generate test programs from the sequences directly. For example, according to the shortest sequence, the test program can be generated as in Table 2. $Check(task_0, task_1)$ function checks the state of $task_0$ and $task_1$ for every step whenever the SVC operation finishes, and gives report if difference is detected. For the example in Table 2, $case\ 1$: $Check(task_0, task_1)$ will check if $task_0$ is $running$, and $task_1$ is $ready$ after SVC $task_0\ activate\ task_1$.

The test program generation benefits from model checking because the optimal paths for all the cases can be found automatically. The test program generation will demonstrate significant power when the test suite is large.

```
void TASK_OsTask0 (void)          void TASK_OsTask1 (void)
    switch(TestIndex)                 switch(TestIndex)
        case 1 :                          case 3 :
            Activate(task1);                  Chain(task0);
            Check(task0,task1);               Check(task0,task1);
            TestIndex++;                      TestIndex++;
            break;                            break;
        case 2 :
            Chain(task1);
            Check(task0,task1);
            TestIndex++;
            break;
```

TABLE 2. Example of test program

5.4. Bug analysis

The entire test cases generated by our method include not only the simple cases, but also the complicated cases when tasks interact with each other running concurrently, which are essential to find the potential bugs. When bugs exist, the user can analyze them with a trace along the test sequence step-by-step as shown in Table 2.

Bug analysis can also be carried out to simulate an existing execution sequence. For example, the RTOS development team reported a bug where a task could be terminated but remained in a ready queue, after $task_1\ activate\ task_3$ and then $task_3\ activate\ task_0$. We simulated the test sequence and a difference was reported.

$$T_1\ activate\ T_3 : T_0.sus, T_1.run, T_3.sus \rightarrow T_0.sus, T_1.run, T_3.run$$
$$T_3\ activate\ T_0 : T_0.run, T_1.ready, T_3.run \rightarrow T_0.run, T_1.ready, T_3.run$$

Following analysis, $task_3$ became $running$ after the operation $task_0\ activate\ task_3$, and $task_0$ became $running$ after the operation $task_3\ activate\ task_0$. The state of $task_1$ was preempted to $ready$. Although the program terminated $task_1$ and forced $task_1$ to the $suspended$ state, the task's identity incorrectly remained in the ready queue.

Because the test can only report where the bugs happen but not the reason, the efficiency of the debug greatly improves if an optimal path from the model checker is provided step by step for analysis.

6. Experiments

Promela code for the model was about 1500 lines. There are 160 lines of java program for test suite generator, and 170 lines of java program for test program generator. The experimental environment was as follows. Model checker: SPIN 5.2.5; RTOS: AUTOSAR 4.0 multicore RTOS; CPU: Intel p8600 2.40GHz; 2.93 GB RAM. The tested SVCs were $activateTask, terminateTask, chainTask$.

Achievement of the heuristic set containing 170 thousands test cases took about 24 hours. Classified with the criteria that did not consider task identities ($criteria_1$), we obtained 33 cases, although OZEK test plan provided 13 cases in [13] and the software development team tested 10 cases. Our 33 test cases included the above mentioned 13 cases and 10 cases. User can also define the other criteria. For example in our experiment we defined another criteria ($criteria_2$) which took into account task identities additionally on $criteria_1$, 684 cases were achieved. With this grained test suite, different task identities were treated as different cases. Because our test process is automatic, a large amount of test cases is also feasible. Test program generation cost an average of about 8.5 minutes for one case.

Figure 9 shows the experiment results of a task management test suite for the system configured as Table 1 with $criteria_1$ (gray lines represents the same cases with OZEK test plan, black lines represents the new cases we found). We used the same test case identities with the OZEK test plan for the same cases, and $1', \ldots, 20'$ for the new cases found by our method.

From this result, 20 new cases were found, containing almost all cases when the executor and affected tasks were on different cores, and several complicated cases when the executor and affected tasks were on the same core, but these were overlooked on the OZEK test plan. Figure 10 shows a typical new case $5'$ in which a multi-activated task chained the higher priority task, which resulted in the high priority task became $running$ and the executor task became $ready$. Figure 11 shows another typical new case $6'$ in which a task chained the task on the other core, resulting in the status changes of four tasks; the executor task became $suspended$ and the $ready$ task on the same core became $running$. On the other core, the affected task with higher priority became $running$ and preempted the $running$ task to $ready$ state.

7. Contributions and future research

Several original characteristics make the work practical and original for industry use. It can significantly improve quality, cut costs, and shorten the time of the test by overcoming the shortages of the conventional test.

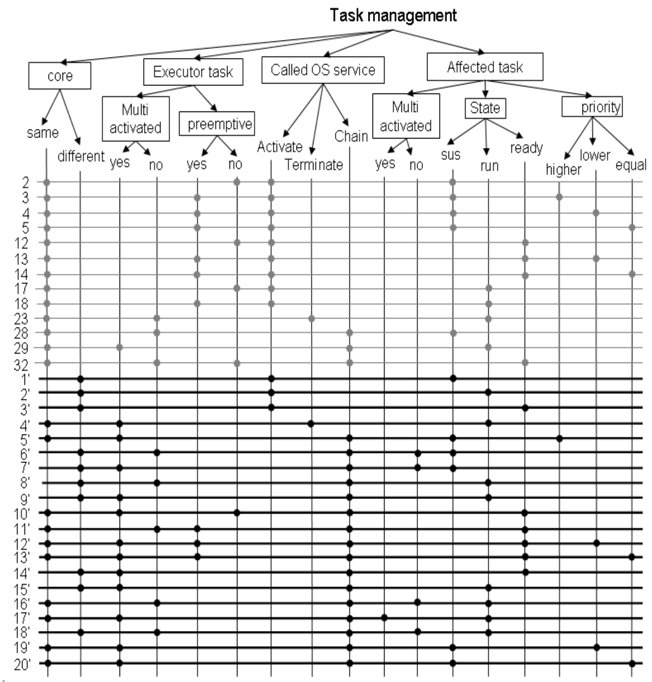

Fig. 9. Task management test plan

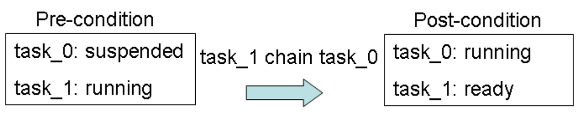

Fig. 10. Example of a typical new test case $5'$ (the executor task and affected tasks are on the same core)

The first and most important contribution involves the original formal model specialized for the test of AUTOSAR multicore RTOS. An abstract model is constructed with the configuration as the initial state. The abstract model can be reused for any AUTOSAR multicore RTOS just with a configuration file for the different system. The new model does not need to be designed whenever the system configuration changes. The target model is concrete; therefore, the test

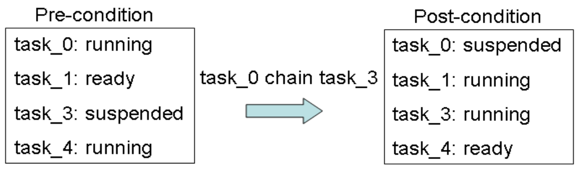

Fig. 11. Example of a typical new test case $6'$ (the executor task and affected tasks are on the different cores)

cases and sequences from the model are executable. It is not necessary to map test cases and sequences expressed at the level of the abstract model to the executable test cases and sequences.

The definition of the critical section greatly suppressed state space, which substantially decreases the possibility of explosion. Moreover, because the states with which the test cases are concerned are the ends of critical sections, the extraction of test cases considering not only the input domain but also the output domain becomes easier than checking the LTL formula.

The whole process of test cases from the extraction of a practical and entire test suite to the generation of the executable test programs is fully automated with model checking or the model checker. The analysis for the bug is easy with the trace from the model checker.

The test cases for a multicore AUTOSAR RTOS are large and complicated, including alarm, event, timing, interrupt, etc. As our project is ongoing, one part of our future research is to complete the model for timing features. A smart performance test is desired by industry, which means the system should run executable test sequences continually, thus covering as many test cases as possible. In other words, with the guidance of the input of test sequence, the system can run continually for a long term of time, for example 24 hours or more. We plan to use model checking to find executable sequences for smart performance testing.

Acknowledgment

The authors would like to thank Tatsuaki Watanabe and Seiji Komori of TOSHIBA Semiconductor Company for their comments on the early work, and also thank Ayumi Kataoka of WITZ Co., Ltd for his kind technical support about RTOS.

References

[1] AUTOSAR: http://www.autosar.org.

[2] Q. Li and C. Yao: Real-Time Concepts for Embedded Systems. CMP Press, 2003.

[3] G. Fraser, F. Wotawa and P. E. Ammann: Testing with Model Checkers: A survey. SNA TECHNICAL REPORT, 2007.

[4] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, H. Zedan: Using Model Checking to Generate Tests from Specifications. ACM Computing Surveys, Vol. 41, No. 2, pp. 1–76, 2009.

[5] B. Lindström, P. Pettersson and J. Offutt: Generating Trace-Sets for Model-based Testing. *18th IEEE International Symposium on Software Reliability Engineering*. IEEE Computer Society, pp. 171-180, 2007.

[6] G. Hamon, L. d. Moura and J. Rushby: Generating Efficient Test Sets with a Model Checker. *2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'07)*, IEEE Computer Society, pp. 261–270, 2007.

[7] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha: Automatic Test Generation from State charts Using Model Checking. *In Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software*, volume NS-01-4 of BRICS Notes Series, pp. 15–30, 2001.

[8] P. E. Ammann, P. E. Black and W. Majurski: Using Model Checking to Generate Tests from Specifications. *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, IEEE Computer Society, pp. 46–54, 1998.

[9] E. Clarke, O. Grumberg and D. A. Peled: Model Checking. The MIT Press, 2000.

[10] F. Ferreira, L. Neves, M. Silva and P. Borba: TaRGeT: a Model Based Product Line Testing Tool. *in CBSoft Tools Session*, pp. 1–4, 2010.

[11] Sal-atg: http://www.csl.sri.com/users/rushby/papers/salatg.pdf.

[12] SpecExplore: http://research.microsoft.com/en−us/projects/specexplorer.

[13] OZEK test plan: http://portal.osek-vdx.org/files/pdf/modistarc/ostestplan20.pdf.

[14] M. Daum, N. W. Schirmer and M. Schmidt: Implementation Correctness of a Real-Time Operating System. *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, pp. 23–27, 2009.

[15] D. Déharbe, S. Galvão and A. M. Moreira: Formalizing FreeRTOS: First Steps. *Brazilian Symposium on Formal Methods (SBMF'09)*, pp. 101-117, 2009.

[16] H. Kuruma, S. Nakajima: Software Development with B (in Japanese). Kindai kagaku sha Co.,Ltd Press, 2007.

[17] G. J. Holzmann: The Model Checker SPIN. *IEEE Transactions on Software Engineering*. Vol. 30, No. 6, pp. 626–634, 1989.

[18] M. C. Gaudel: Testing Can Be Formal, Too. *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT'95)*, Springer-Verlag, pp. 82–96, 1995.