Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 14, no. 6, June 2018, pp. 1241-1250
DOI: 10.23940/ijpe.18.06.p15.12411250

Test Scenario Generation using Model Checking

Zhixiong Yin[a], Min Zhang[b], Guoqiang Li[c], and Ling Fang[d,*]

[a]*School of Engineering Sciences, University of Chinese Academy of Sciences, Beijing, 100049, China*
[b]*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, 200062, China*
[c]*School of Software, Shanghai Jiao Tong University, Shanghai, 200240, China*
[d]*Institute of Technology Innovation, Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei, 230031, China*

Abstract

Testing, including designation, execution and bug analysis has been broadly adopted in various industries. Test cases must be designed to confirm the entire behavior of the object system. In practice, a test case normally includes a series of operations that cold conduct the system status eventually to fulfill the precondition of the targeted test case. However, the applications to trigger the function calls of the software system would often be manually composed in traditional test activities, which is difficult especially for complex concurrent systems. Insufficient testing might result in hidden system defects, which increases the likelihood of economic loss or human injury. Model checking is a formal technique that can check the properties of a system automatically with strictness, completeness, and traceability. In this paper, a novel formal approach is proposed to systematically generate test scenarios automatically with traceability by the model checking technique. Meanwhile, the scenarios are reliable and precise, and debugging also becomes easier. Moreover, despite the reduction in demand of an experienced test engineer to design the target test cases, the coverage could be improved as the tedious and complicated processes of test scenario generation are accomplished by the model checker. This method has been applied in two practical case studies, and the results show the effectiveness of the proposed approach in terms of high coverage, automation, traceability and reusability.

Keywords: test scenario; model checking; model-based testing

(Submitted on March 17, 2018; Revised on April 14, 2018; Accepted on May 21, 2018)

1. Introduction

Nowadays, the increasing complexity of software systems creates new challenges for safety, where quality and reliability are highly required, but are difficult to be ensured. Testing is a necessary way to ensure the correctness of software systems in practice and to monitor if systems operate properly or not [1,9]. Written test programs should include a description of the functionality that needs to be tested. Preparation is required to ensure that the test can be conducted, which implies a series of executions from the initial status until the system reaches the precondition, i.e., a system under test (SUT) [1,9]. Traditionally, manual testing is not only time-consuming and costly, but also requires technical expertise, especially in case the system is complicated with concurrency.

Model checking is a formal technique for assessing functional properties of systems [3,4]. Model checking requires a model of the system under consideration, a desired property and systematically checking whether or not the given model satisfies this property. Model checking is strict, complete, and automatic: strictness refers to the assessment as implicit and precise; completeness means the inspection covers the integral model; automation provides the convenience as it is without manual intervention. Model checking provides traces when it performs the inspection or verification. The main contribution of this work is systematic generation of test scenarios using model checking. Meanwhile, model checking also brings benefits for test coverage, test reusability, and system debugging.

* Corresponding author.
E-mail address: fangl@hfcas.ac.cn

Our method aims for continuous and traceable testing with high coverage that is valuable for systems with high requirements, complicated concurrency, or costly reboots. Considering a serial test input representing a continuous behavior of an object system that contains complicity concurrency features is not only time-consuming and costly, but also requires technical expertise and is almost impossible as high coverage is required. Additionally, each test case must be executed on the SUT where the previous test input triggers a system change. Test scenarios are generated automatically by the traceability of model checking.

The coverage is achieved along with automation and traceability because assigning function points must be taken into account and applied for test operations. Moreover, debugging is highly complicated in traditional methods, and testing engineers have to speculate a potential reason from an unexpected phenomenon. In addition, bugs might be overlooked if abnormalities are failed to be observed. In our approach, debugging becomes easier because model checking provides the trace of each test case and the bugs can be quickly located by comparing the execution of system with the trace. Reusability a is naturally complied advantage because when systems change or updates are required due to further development or maintenance, the revision of the model and re-generation of testing would be uncomplicated.

The structure of the remaining part of this paper is as follows: Section 2 explains the main concepts of model checking and testing related to our work; Section 3 describes how the method was approached, and Section 4 demonstrates two running examples; Section 5 introduces the related works in detail; Section 6 concludes the whole paper.

2. Preliminaries

The basic idea of the approach is to build a model that conforms to the requirements of the object system, and this model can conduct the system along the functional properties of systems continuously with the traceability of model checkers. The following paragraphs present the concept of model checking and testing related to our work.

2.1. Model checking

Model checking is a formal technique for automatically verifying properties of finite-state systems and establishing the correctness, performance, and reliability of software systems [3, 4]. Model checking can be rephrased as a decision problem as follows. Given a finite model M and a state formula Φ for a state s in M, determine whether $s|=_M\Phi$. The SPIN model checker [7] is a prominent example of an automated verification tool such as that used in our work. The SPIN modeling language is Promela, and the linear temporal logic (LTL) is the specification logic, which is based on a linear time perspective and is used to formalize temporal properties of systems.

Definition 1 (Kripke structure). A Kripke structure is a 4-tuple $K=〈S_0,S,R,L〉$, where S is a set of states that contain the initial state s_0, R is a transition relation between states, and L is a labeling function that maps from states to sets of atomic propositions, i.e., $L:S\rightarrow 2^{AP}$, where AP denotes a set of atomic propositions. $L(s)$ is the set of all the atomic propositions in AP that hold in the state s.

A path denoted by π (probably finite or infinite) describes one possible execution of K and is defined as a sequence of states $s_0, s_1,...$ such that $\forall i\geq 0.\Delta(s_i, s_{i+1})>0$. Let $\Pi_{K,s}$ denote the set of all the paths of K starting from *state* s, π_i denote the i position in the path π $(i\geq 0)$. For $s, s'\in S$, the length of a path from s to s' is denoted as $D(s, s')$, and the shortest path among $\Pi_{K,s}$ is denoted as $Ð(s, s')$ [3].

Definition 2 (Linear Temporal Logic, LTL).

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

$\phi ::= true|a|\neg\phi|\phi_1\wedge\phi_2|\circ\phi|\phi_1 U\phi_2.$

LTL Semantics: The semantics of LTL formulas can be defined in terms of Kripke structure. We use $\pi|=\phi$ to denote that a path π satisfies an LTL formula ϕ, which is defined inductively as follows:

$\pi|= a$ *iff* $a\in L(\pi_0)$
$\pi|\neq \neg\phi$ *iff* $\pi|\neq\phi$
$\pi|=\phi_1\wedge\phi_2$ *iff* $\pi|=\phi_1$ *and* $\pi|=\phi_2$
$\pi|=\circ\phi$ *iff* $\pi_1|=\phi$

$\pi \models \phi_1 \cup \phi_2$ *iff* $\exists j \geq 0.\ \pi[j\ldots] \models \phi_2$ *and* $\pi[i\ldots] \models \phi_1$, *for all* $0 \leq i < j$.

The LTL formula ϕ that holds in the path π of K is represented as $K,\pi \models \phi$. In particular, we use $K \models \phi$ to denote that all paths that start the initial state of a Kripke structure K satisfies ϕ, i.e., $K \models \phi$ if and only if for all $\pi \in \Pi_{K,s0}$ there is $\pi \models \phi$, where s_0 is the initial state of K.

2.2. Concept of tests

The related concept of testing includes various contents such as test case, test scenario, test suite and system under test SUT [9].

Definition 3 (Test case). A test case tc_i is a triple $\{S_{pre}, \Omega, S_{post}\}$, consisting of a specification of the inputs S_{pre}, testing procedure Ω, and expected results S_{post}

A test case defines a single test to verify compliance according to a specific requirement. For applications or systems without formal requirements, test cases can be written based on the accepted normal operation of programs. A test case is the basic concept in testing technique that is constructed to simulate the execution of end-user applications. A set of test cases is called a test suite.

Definition 4 (Test sequence). A test sequence $seq(s_0, tc)$ for test case tc is a finite path $s_0, s_1, \ldots, s_n$ of a system where s_0 is an initial state of the system and s_n is a state satisfying the input specification of tc.

In practical testing, hypothetical stories are used to help the tester think through a complex problem or system, and they finally reach the system status where the target test case can be executed. A test sequence might traverse several intermediary system states before arriving at the pre-condition $s(i)_{pre}$ of objective test case tc_i ($\{s_{(i)pre}, \Omega_i, s_{(i)post}\}$).

Definition 5 (Test scenario). A test scenario ω is a finite path that traverses more than one objective test case. The test scenario $\omega(s_0, seq_{tc0}, seq_{tc1}, \ldots, seq_{tcn})$ starts from the initial status s_0 and traverses $tc_0, tc_1, \ldots$.until it arrives at tc_n.

Each pair of adjacent test cases tc_i, tc_{i+1} is connected by a test sequence seq_i, starting from the terminate state of $s_{(i)post}$ of tc_i, and ending at $s_{(i+1)pre}$ of tc_{i+1}. When there is only one assigned test case, the test scenario degenerates into one test sequence. The scenario ω is a concept of testing that represents a continuous operation of testing. In this work, ω is solved with the path π of model checking.

The difference between test sequence and test scenario must be clarified. Given a test case as the assertion property, a test sequence is generated that contains the execute trace from the start point of system status to the targeted case. When given more than one test cases, model checking is processed to find the test sequences that can continuously executed. The path jointing test sequences that can conduct the system and execute along the given test cases is defined as the test scenario.

2.3. Model-based testing

Model-based testing is a technique for designing system testing. Models are specified to represent the desired behavior of a system under test (SUT), or to represent testing strategies and a test environment. Then, tests are derived from the models in different ways. Because test suites are derived from models and not from source code, model-based testing is usually used as black-box testing. For example, to ensure that the system is behaving in the same sequence of actions derived from the model.

There are various known ways to deploy model-based testing, e.g., model checking, theorem proving, finite state machine, etc. In this paper, model checking is adopted as the automation it offers. Tests can be derived mechanically. When used for testing, a model of the system under test and a property to test are provided to the model checker, and the model checker detects witnesses and counterexamples. A witness is a path where the property is satisfied, whereas a counterexample is a path in the execution of the model where the property is violated. These paths can again be used as test cases, test sequences, or test scenarios.

The essential features of model checking are strictness, heuristics, and automation. Model-based testing brings several benefits, such as a higher level of automation, possibility of exhaustive testing, and reusability of tests. On the other hand, model-based testing requires a formal specification or model to carry out testing, which is not easy in practical development. Moreover, test cases are tightly coupled to the model.

3. Test scenario generation using model Checking

Figure 1 shows a process overview of the test method. The process is formed with sequenced stages: model construction, including verification of the model that guarantees the test scenario generation is based on a correct model; computing the trace for every pair of adjacent test cases; combining the test sequences and generating a scenario that traverses all the assigned test cases. A model contains the space of states and transitions as shown in Figure 2, e.g., system transits from the state “aa” to “bb” when the condition “cc” is satisfied.

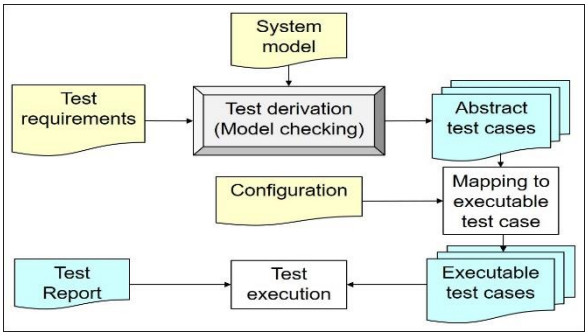

Figure 1. Process outline

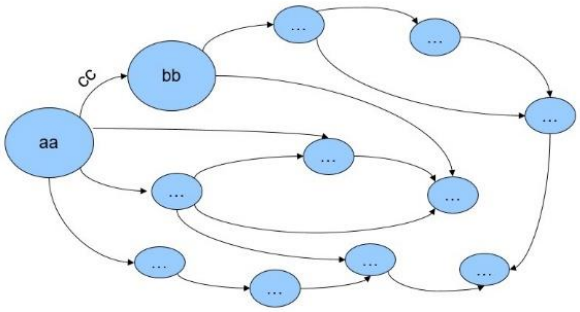

Figure 2. Model space

3.1. Modeling

Modeling is the crucial step for model-based testing. In fact, the correctness of the model is the foundation of testing as all consequences in the testing process would rely on the constructed model.

Model construction. In our work, the model would be composed of two parts for practical convenience: the system specification on the abstract level, which should not be frequently changed, and configuration that might be revised along with the project development/maintenance or environmental variety. The model on the abstract level can be configured to a concrete model corresponding to an application prospect. For example, “*an automobile at an entrance of a car park may enter if the parking area is available, and return to the road if not*” is on the abstract level. “*the automobile with plate number A12345 at the entrance 5 of the car park may enter and take vacant lot 3, and return to the road if not*” is on the concrete level.

There is another situation where the system requirements could be described as system specification, and the transitions and constrain relations were defined on a sugar language, i.e., a description of model that is more abstract and easy to be understood, and the primitive parts must be substituted with Promela language. For example, as shown in Figure 2, “*state aa transits to state bb when condition cc is true*” is the system specification. Then, the sugar is concretely formulated as “*state aa: a car on road; state bb: a car at entrance; condition cc: the parking is available*”.

For model-based testing, the model is constructed manually according to the system specification and the requirements of the object system, i.e., the substantial part of the model is not changed frequently. The model must be configured to correspond to the testing program that consists of a list of subroutines. The modeling is the most complicated phase of model-based testing, as it requires not only understanding of the system requirements, testing objectives, the abstract and concrete demarcations, but also extracting the essential characteristics of the object system related to the test purpose. Most of the existing work generates test cases or scenarios, however the mapping process to the executable programs includes tedious manual work. In this study, test programs are generated from the configured model automatically, which is useful for industry needs.

Model configuration. Test cases derived from a model are functional tests on the same level as the model. When the model is specified on an abstract level, e.g., describing the requirements of a system, the extracted test cases cannot be directly executed against an SUT. Executable test cases need to be derived. This is achieved by mapping the abstract test cases to concrete test cases suitable for execution. In some model-based testing environments, models contain enough information to generate executable test suites directly. In others, elements in the abstract test suite must be mapped to specific statements or method calls in the software to create the concrete test cases. This is called solving the "mapping problem"[1].

Model verification. When a system becomes complicated, the model checker has to be used to guarantee the correctness of model. Besides the defects as dead-lock and live-lock, which could be automatically detected with a model checker, any undesired properties such as reachability, which can be expressed with LTL formulas or assertions, would be identified. The model is revised until the expected results could be output without any error or explosion of any problem as in Figure 3.

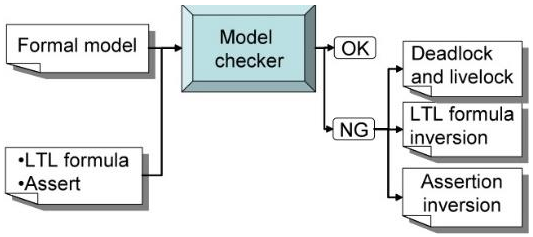

Figure 3. Model verification

3.2. Test scenario generation

For a test suite $TS:\{tc_0, tc_1, \ldots, tc_i, \ldots, tc_n,\}$ and assigned subset $TSS: TSS \subseteq TS$, test scenarios can be generated with these iterated steps until all the test cases in *TSS* have been processed.

- $STEP_1$: Take a test case in *TSS*;
- $STEP_2$: Set pre-condition S_{pre} of this test case as the counterexample, and model checking will find a test sequence between the initial state to S_{pre}. For the firstly taken test case, the initial state equals to the system beginning state S_0;
- $STEP_3$: Execute Ω of this test case and the system status transits to S_{post} of this test case;
- $STEP_4$: Set the post-condition S_{post} of this test case as the initial state;
- $STEP_5$: Delete the processed test case from *TSS*.
- $STEP_6$: If *TSS* is not empty, return to $STEP_2$.

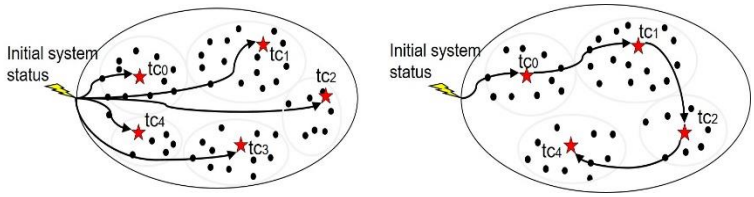

Figure 4. Test sequence (left) and scenarios (right)

Figure 4 (left) shows the three sequences from the initial state to the cases tc_0, tc_2, and tc_4, respectively. This kind of sequence, which generally involves several steps, shows the shortest trace to the target case; therefore, it is convenient for unit tests or debugging analysis. Figure 4 (right) shows the test scenario from the initial state and traverse tc_0, tc_2, and tc_4 continuously. Test scenarios are valuable in two circumstances:

- For the test objectives in which continuous operations are concerned. For example, embedded systems/equipment where a continuous executing program is required to confirm the continuous operations with complicate function modules.
- For systems in which the cost of restarting the system to execute a new test sequence is quite high. For example, testing for aviation or nuclear systems. Continuous test scenarios that provide high coverage will reduce the time and cost.

It is understandable how difficult it was to design a scenario that consists of thousands of steps, and this difficulty becomes prominent for function or performance testing of the embedded systems [5, 11, 14]. This complication has been removed in this paper through model checking, which is helpful in finding the traces from one case to another until all the sequences are finally connected to one continuous test scenario.

The coverage of testing is an important requirement of testing. In this work, heuristic of model checking exerts the advantage of improving the coverage. Given the test cases as the assertion property, the model checker can output the exhaustive test sequence or test scenarios. As the derivation of tests is automatic, once the framework is constructed, the test generation is reusable only with the revision of model.

The length of path to a counterexample reported by a model checker depends on both the algorithm used for state space exploration and the way the property is encoded. There are works that provide algorithms for an automaton to accept the shortest counterexamples [1]. In our work, we applied the function of model checker to find the shortest path (the best route to the target test cases).

4. Running example

This section describes two kinds of typical applications of our method: a parking payment system and a real-time operating system application programming interface (RTOS API).

4.1. Parking payment system

A car parking system is an integrated management system of high quality, high efficiency and high reliability. As the system is developed for each different client or each new function, the test plans have been manually developed for each revision before our work. With our approach, the test generation can be automatically generated, and the cost for testing is greatly reduced.

Modeling: The model is constructed based on an existing system, and the existing test plan is also used for comprehension of the system. The model is specified based on a sugar language and is then mapped to the specification language Promela [7].

Figure 5[1] demonstrates the parking progress such as, a car from the road goes to the entrance and enters if the parking lot is available (*park=F*), but returns to the road if it is unavailable (*park=F*); after a car is allowed to enter, the video surveillance system tries to recognize the number plate and allows parking if it was successful (*vrEn=T*). A temporary access card would be issued if the number plate is failed to be recognized (*vrEn=F*); from the parking lot, the car goes to the road directly when the toll is false (*toll=F)*, i.e., free service.

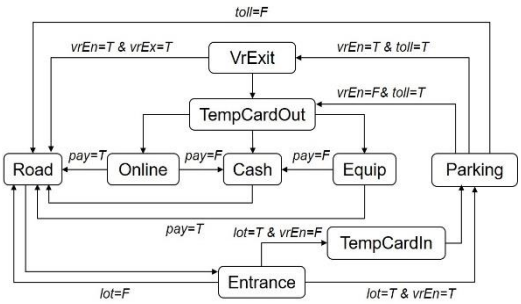

Figure 5. The transition relations of the parking payment system

Table 1 shows the description of sugar (up) and an example (low) (*action* and *condition* are shortened to *act* and *cond; True* and *False* are shortened to *T* and *F*). In this example, $state_1$ has two success states. One is $state_4$, which executes act_7 when $cond_5$ is true. Another success is $state_7$, which executes act_5 on $cond_9$. The states, conditions, and actions are defined with Promela language and are substituted as "*def:* $cond_3$ $a \wedge b$". Table 2 describes the model according to the definition in Table 1.

Test Scenario: Testing for this kind of system concerns continuous operations. The program shown in Table 3 is generated from the model and guides the system to eventually execute the test cases continuously traversing $state_0$, $state_1$ and $state_9$.

[1] The meaning in Figure 5 and Table 2. vrEn: enter by number plate recognition; vrEx: exit by number plate recognition; toll: in time duration that must pay for parking; pay: pay for parking; lot: parking lot is available; TempCardIn: enter with temp card; Entrance: at entrance; Road: on road; Parking: car is parking; Equip: pay by equipment; Cash: pay with cash; Online: pay online; TempCardOut: pay with temp card; VrExit: trying to exit by number plate recognition.

Table 1. Sugar language of specification/*comments*/

SPEC: {R(N,N),R∈(C,A)} /*R is the transition relation of nodes N, N transits to the success node N when condition C is true, and trigger action A. N, C, A are defined according to PROMERA.*/
Definition: *def : cond3 "a∧b"* *def : cond5 "c∧d"* *def : cond9 "a"* *def : act5 "a = false"* *def : act7 "b = true"* *def : act8 "c = true; d = true"* Sugar: *state1 : cond5, state4, act7; cond3, state5, act8; cond9, state7, act5*

Table 2. The model of the parking payment system

Define: *$state_0$: road, $state_1$: entrance, $state_9$: park* *$cond_1$: vrEn = true, $cond_2$: vrEn ∧ vrEx, $cond_9$: free* *act_1 : allow := vr, act_9 : car := road*
Sugar: *state0 : true, state1* *state1 : state1, act1, state9* *state9 : cond9, act9, state2; cond0, act0; state0,* *...*

Table 3. Test scenario of parking payment system

seq_0: $state_0$ to $state_9$ traversing $state_1$ *$step_1$: from road to entrance, the condition is T (on any condition)* *$step_2$: from entrance to park, when vrEn = T (plate number is recognized correctly) and lot = T (park lot is available).*
seq1: state9 to state0 step3: from park, if toll=F (during charge free service), to road

4.2. RTOS API.

The RTOS (real time operating system) is a layered architecture that makes component-based and complicate design possible. RTOS provides the infrastructure and application interfaces for efficient software development and sharing by defining standardized application programming interface (APIs) [14]. Test scenarios are about to generate a program that can be executed in an embedded system. More detailed contents can be referred in our previous work [10].

A task is the basic unit of execution controlled by an RTOS. An application consists of one or more tasks. Each task executes a predefined program with a given priority. A task has the states, i.e., *ready*, *running* and *suspended*. API *activateTask* stipulates the rules to activate a task to the *running* state. Similarly, *terminateTask* 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. The behaviors become confusing with the consideration of constraining conditions of priority strategy [14].

Modeling: An abstract model is constructed manually conforming to the requirements of the objective RTOS standard. The configuration subsumes the concrete information about the tasks identities, priority values, etc. As shown in Table 4, with the description language of the model checker, the abstract model and configuration can be described as functions and parameters, respectively.

Table 4. Abstract (up) and configured model (down)

IF: *eTask act pTask ∧ eTask.prio < pTask.prio* *THEN : eTask.state = ready, pTask.state = run*
IF: *$task_1$ act $task_0$ ∧ $task_1$.prio < $task_0$.prio* *THEN : $task_1$.state = ready, $task_0$.state = run*

The abstract description means that *eTask* (executor task) is preempted to the *ready* state and *pTask* (passive task) transits to the *running* state when *eTask* activates the *pTask* and the priority of *eTask* is lower. The configuration $task_0$–$task_2$ share $core_0$, and $task_3$–$task_5$ share $core_1$, and all the tasks are preemptive.

Table 5. Typical test cases (/*comments*/; test case is shorten as "tc", "Task" is shorten as "t")

tc_0: t_0 act t_1 : t_0.run, t_1.sus, t_3.run, t_5.sus → t_0.run, t_1.ready, t_3.run, t_5.sus /*a task activates another task on the same core, and makes it ready*/
tc_1: t_3 act t_5 : t_0.run, t_1.sus, t_3.run, t_5.sus → t_0.run, t_1.sus, t_3.run, t_5.ready /*a task activates another task on the same core, and makes it ready*/
tc_2: t_1 act t_3: t_0.sus, t_1.run, t_3.sus, t_5.sus → t_0.sus, t_1.run, t_3.run, t_5.sus /*a task activates another task on a different core, and makes it running*/
tc_3: t_1 act t_3: t_0.sus, t_1.run, t_3.sus, t_5.run → t_0.sus, t_1.run, t_3.run, t_5.ready /*a task activates another task on a different core, makes it running. The running task on the different core is preempted */
tc_4: t_3 tmn : t_0.sus, t_1.sus, t_3.run, t_5.sus → t_0.sus, t_1.sus, t_3.sus, t_5.sus /*a task terminates itself*/
tc_5: t_3 tmn : t_0.run, t_1.sus, t_3.run, t_5.ready → t_0.run, t_1.sus, t_3.sus, t_5.run /*a task terminates, and the ready task on the same core becomes running*/

Formulas 1-3 are examples of LTL formulas (i, j are free variables that must be bound to the identities of tasks; n is the number of tasks on the CPU). Formula 1 requires the property that the priority of $task_i$ is higher than that of $task_j$ and $task_j$ is preemptive, but $task_i$ is ready and $task_j$ is *running* ($i \neq j$). This property represents a defect of priority violation. Formula 2 represents an incorrect state, namely that there is a *ready* task, where the CPU is not occupied by any task. Formula 3 represents another incorrect state, namely that there is more than one *running* task.

$$\neg(task_i = ready \wedge task_j = running \wedge (priority_i < priority_j)) \quad (1)$$

$$\neg \left(\sum_{i=0}^{n} taski = ready > 0 \wedge CPU = free\right) \quad (2)$$

$$\neg \left(\sum_{i=0}^{n} taski = running > 1\right) \quad (3)$$

Test Scenario: Table 5 shows several typical test cases that can be extracted from the model in Table 4. For example, tc_0 is a case where task$_0$ activates $task_1$. As the priority of tc_0 is higher, tc_1 enters the ready state and waits for the tc_0 stops. The program shown in Table 6 guided the system to eventually execute the test cases continuously traversing tc_2 and tc_5 within the executable environment.

When there are thousands or more test cases for a practical system, automatic generation of test scenarios for any assigned cases, which are newly added or modified functions, is very valuable. Until this work, this scenario was designed manually. For a real industrial system, since the test scenarios become much more complicated to design, test engineers usually give up on complex testing, which causes potential ignorance of system defects.

Table 6. Test scenario traversing tc_2 and tc_5

seq_0 : s_0 to tc_2
(t_0 act t_1) : t_0.run, t_1.sus, t_3.run, t_5.sus → t_0.run, t_1.ready, t_3.run, t_5.sus
(t_0 tmn) : t_0.run, t_1.rdy, t_3.run, t_5.sus → t_0.sus, t_1.sus, t_3.run, t_5.sus
(t_1 act t_3) : t_0.sus, t_1.run, t_3.sus, t_5.sus → t_0.sus, t_1.run, t_3.run, t_5.sus
seq_1 : tc_2 to tc_5
(t_3 act t_5) : t_0.sus, t_1.run, t_3.run, t_5.sus → t_0.sus, t_1.run, t_3.run, t_5.rdy
(t_3 tmn) : t_0.sus, t_1.run, t_3.run, t_5.rdy → t_0.sus, t_1.run, t_3.sus, t_5.run

Table 7. Pseudo program

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

```
void TASK_OsTask1 (void)
    switch(TestIndex)
        case 3 :
            Activate(task3)
            Check(task0,task1);
            TestIndex++;
            break;
```

4.3. Analysis and discussion

The following is the analysis and discussion about our method through the above running examples from the aspects of efficiency and the ability to debug, which are the most concerning issues.

Efficiency: Automatic test generation made rapid development possible. During the development of the parking payment system, after the test scenario generation system was integrated with JUnit, test engineers were almost allayed from tedious and time cost test work. The coverage was also improved as assigning desired cases became much easier. Manual generation for each system revision costs around 1 week for 198 cases. 291 cases were generated by our method including 93 new cases. Additionally, time and engineering resource costs were also saved due to the automation.

For the project of RTOS, our method also played an important role. Compared to the OZEK test plan, which provided 13 cases in [13], in which the test engineer must compose test programs for each round of system revision for several days, 33

test cases in this study including the above mentioned 13 cases were found. In addition, a pseudo test program can be generated automatically as well, which only needs a very light modification.

However, the modeling requires a highly knowledgeable engineer, e.g., the comprehension of the object system as well as the skill to construct a formal model and use a model checker. Moreover, the explosion problem of model checking also obstructs the application of the method for a large scale system [4].

Bug detection: Because of the automation and improved coverage, potential bugs could be found, and debugging became much easier because the tester can analyze them with a trace step-by-step. The parking payment system is an existing system that has been developed for several years. Revisions with additional new functions in the system are under development. After modeling and test case designing with JUnit, bugs were found along with the system development and maintenance. For example, we found several payment errors that did not comply with the strategy.

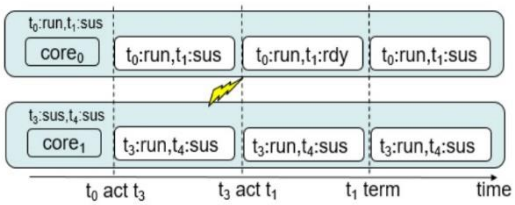

Figure 6. The bug of RTOS API

For RTOS API, bugs were reported as shown in Figure 6, where a task could be terminated but remain in a *ready* queue after $task_1$ activated $task_3$ and then $task_3$ activated $task_0$. After the simulation of the test scenario, a bug was reported, which was caused by an ignored state transition triggered by concurrency.

5. Related works

Model-based testing has been investigated for several decades. Despite many works described in references [2, 6, 8], it has been demonstrated that formal methods could contribute greatly in this field, but many problems still remain unsolved. The key technique of the method in this paper is to find traces to connect all the assigned test cases continuously.

Heimdahl et al. [12] proposed that test cases can be treated as a counterexample to find a test sequence. In this domain, test cases providing common coverage seem to be quite short, thereby allowing bounded model checkers to perform very well. There are also existing tools such as ZIPC [18] and specExplore [15] that can generate a test sequence for unit testing. The previous work [10] also proposed a system for unit testing of the RTOS. However, the problem of generating a scenario that could traverse arbitrary states remains unsolved.

With semi-formal specification language, Toppers project [17] reported a test sequence generation based on 59,000 lines of manually designed code for the multi-core RTOS. SLAM [16] was a Microsoft project designed to verify the Windows kernel. Model checking was used to show the correct API usage in device drivers, and the verification objective was a large-scale commercial software. The drivers used all the features of the C language and ran in a very complex environment, which made the analysis challenging. SLAM was a large and costly project that took a long time. In comparison with previous efforts, the method in this paper is easy to apply and much less expensive.

This method, as described in this paper, can provide arbitrarily assigned test scenarios according to the user's purposes. Therefore, the cost of testing can be significantly reduced because instead of testing thousands of programs, a few test programs that include many cases could achieve the same effectiveness without the restarting phase for each case. The tedious process of considering the scenarios to test an objective case is saved. In another circumstance, the model for test generation is configured to the environment. Thus, the mapping process from the abstract level to an executable program is avoided.

6. Conclusions

Testing plays an essential role to ensure the quality of system development. Manually designed test programs can only provide low coverage and low efficiency due to the high cost in time and resources. Moreover, defects remain undiscovered. This work solves several problems within industrial practices in an innovative way by using model checking.

The main contribution of this work is test scenario generation with the advantages of automation, strictness, completeness, and traceability. The test scenarios are able to traverse assigned test cases according to the user's purposes. Model checking can output exhaustive test cases that imply high coverage of testing. The sequences of testing are generated automatically, extricating the test engineer from tedious and complicated work to design a path to execute the test cases. For arbitrarily assigned test cases, test scenarios will be generated to traverse them, which is valuable for embedded systems where the continuous operation is concerned, or for systems such as aviation where the reboot is costly. The debugging process becomes convenient via the step by step diagnostic. The test program generator can be reused for different test purposes or different environments. When our method was applied in two practical industrial projects, the results show that this method is feasible and practical to reduce time and cost of resources. This method is expected to improve the quality of systems.

References

1. F. Abbors, and D. Truscan. "Approaching Performance Testing from a Model-Based Testing Perspective", Proceedings of the 2th International Conference on Advances in System Testing and Validation Lifecycle (VALID), pp.125–28, 2010.
2. C. Baier, and J. P. Katoen. "Principles of Model Checking". Publisher: The MIT Press, ISBN-10: 026202649, 2008.
3. T. Ball, and S. K. Rajamani. "SLIC: A Specification Language for Interface Checking". Technical Report, MSR-TR-2001-21, Microsoft Research, 2002.
4. B. Beizer. "Software Testing Technologies". ISBN: 1850328803, Publisher: Itp-Media, 2nd Edition 1990.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. "Model Checking", Publisher: MIT Press, ISBN: 9780262032704, 2000.
6. M. Daum, N. W. Schirmer and M. Schmidt. "Implementation Correctness of a Real-time Operating System". Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods. IEEE Computer Society, pp.23–27, 2009.
7. G. Fraser, F. Wotawa, and P. E. Ammann. "Testing with Model Checkers: A survey". SNA Technical Report, 2007.
8. L. Fang, T. Kitamura, T. B. N. Do, and H. Ohsaki. "Formal Model-Based Test for AUTOSAR Multicore RTOS". Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation. IEEE Computer Society, pp.251–259, 2012.
9. J. M. Glenford. "The Art of Software Testing". Publisher: Wiley, ISBN: 0471469122, 2004.
10. G. Hamon, L. D. Moura, and J. Rushby. "Generating Efficient Test Sets with a Model Checker". Proceedings of the 2th IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society, pp.261–270, 2007.
11. M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. "Auto-generating Test Sequences Using Model Checkers: A Case Study". Proceedings of the 3th International Workshop Formal Approaches to Software Testing. Springer, LNCS 2931, pp.42–59, 2004.
12. G. J. Holzmann. "The Model Checker SPIN. IEEE Transactions on Software Engineering". Vol. 30, No. 6, pp.626–634, 1989.
13. Q. Li, and C. Yao, "Real-time Concepts for Embedded Systems", CMP Press, 2003.
14. B. Lindstrom, P. Pettersson, and J. Offutt. "Generating Trace-Sets for Model-based Testing". Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering. IEEE Computer Society, pp.171–180, 2007.
15. OSEK test plan: portal.osek-vdx.org/files/pdf/modistarc/ostestplan20.pdf.
16. SpecExplore homepage: research.microsoft.com/en-us/projects/specexplorer.
17. Toppers homepage: www.toppers.jp/en/index.html.
18. Zipc homepage (in Japanese): http://www.zipc.com/download/catalog/pdf/product/zipc tester.pdf.