SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

May 2018, Vol. 61 052102:1–052102:15
doi: 10.1007/s11432-016-9039-4

autoC: an efficient translator for model checking deterministic scheduler based OSEK/VDX applications

Haitao ZHANG[1*], Zhuo CHENG[2], Guoqiang LI[3] & Shaoying LIU[4]

[1]*School of Information Science and Engineering, Lanzhou University, Lanzhou* 730000*, China;*
[2]*School of Information Science, Japan Advanced Institute of Science and Technology, Nomi* 923-1211*, Japan;*
[3]*School of Software, Shanghai Jiao Tong University, Shanghai* 200240*, China;*
[4]*Faculty of Computer and Information Sciences, Hosei University, Tokyo* 184-8584*, Japan*

Received 3 November 2016/Revised 21 January 2017/Accepted 20 February 2017/Published online 11 August 2017

Abstract The OSEK/VDX automotive OS standard has been widely adopted by many automobile manufacturers, such as BMW and TOYOTA, as the basis for designing and implementing a vehicle-mounted OS. With the increasing functionalities in vehicles, more and more multi-task applications are developed based on the OSEK/VDX OS. Currently, ensuring the reliability of the developed applications is becoming a challenge for developers. As to ensure the reliability of OSEK/VDX applications, model checking as a potential solution has attracted great attention in the automotive industry. However, existing model checkers are often unable to verify a large-scale OSEK/VDX application that consists of many tasks, since the corresponding application model too complex. To make existing model checkers more scalable in verifying large-scale OSEK/VDX applications, we describe a software tool named autoC to tackle this problem by automatically translating a multi-task OSEK/VDX application into an equivalent sequential model. We conducted a series of experiments to evaluate the efficiency of autoC. The experimental results show that autoC is not only capable of efficiently sequentializing OSEK/VDX applications, but also of improving the scalability and efficiency of existing model checkers in verifying large-scale OSEK/VDX applications.

Keywords OSEK/VDX application, deterministic scheduler, software formal method, model checking, sequentialization

Citation Zhang H T, Cheng Z, Li G Q, et al. autoC: an efficient translator for model checking deterministic scheduler based OSEK/VDX applications. Sci China Inf Sci, 2018, 61(5): 052102, doi: 10.1007/s11432-016-9039-4

1 Introduction

Developments in the automotive industry and electronic technology increasingly involve the deployment of vehicle-mounted systems in automobiles. However, reusing and transplanting the developed systems have become a serious problem for automobile manufacturers, since there is no uniform development standard in the automotive industry. To overcome this problem, European of automobile manufacturer association develops and promulgates an automotive OS standard named OSEK/VDX [1] in 1994. The standard has now been widely adopted by automotive manufacturers, such as BMW, Opel and TOYOTA. Currently, more and more complex applications have been developed based on the OSEK/VDX OS.

As shown in Figure 1, an application developed to run on OSEK/VDX OS consists of multiple tasks, and these tasks are concurrently executed under the scheduling of OSEK/VDX OS scheduler (a deterministic

* Corresponding author (email: htzhang@lzu.edu.cn)

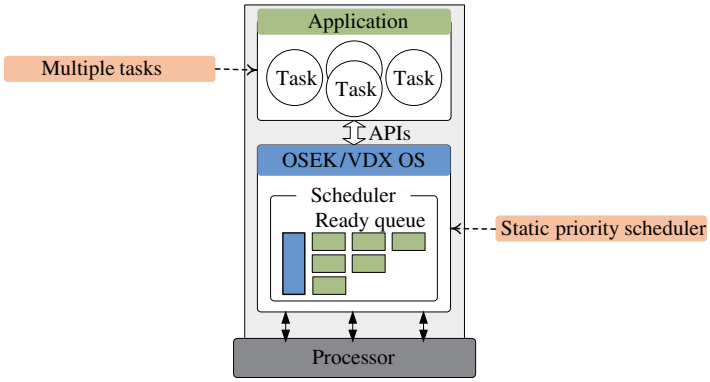

Figure 1 (Color online) Structure of OSEK/VDX application.

scheduler called static priority scheduler is adopted by OSEK/VDX OS to dispatch tasks, where a ready queue is used to manage the scheduling order of tasks). Moreover, tasks within an application can invoke application interfaces (APIs) to dynamically change the scheduling order of tasks, e.g., activate a higher priority task. However, developers face the challenge of ensuring the reliability of developed OSEK/VDX applications which are becoming increasingly complex because of the concurrency of tasks and the dynamic scheduling. To ensure the reliability of developed OSEK/VDX applications, model checking [2–4] as an automatic and exhaustive checking technique has attracted great attention in the automotive industry.

Recently, the persistent improvement of model checking techniques has resulted in many different model checking techniques based automatic model checkers such as Spin [5], ESBMC [6], and VCC [7] being built and widely using to verify concurrent programs [8, 9]. To make existing model checkers such as Spin model checker successfully verify OSEK/VDX applications, Refs. [10, 11] have shown a plain method. In the proposed method, all tasks within an application are simulated by concurrent `process`es. Moreover, an application is explicitly verified by adding a special concurrent `process` for OSEK/VDX OS to realize deterministic scheduling behaviours and respond to the APIs invoked from tasks. Although this method can be applied to verify OSEK/VDX applications, the method is not capable of verifying a large-scale OSEK/VDX application that contains many tasks, as reported in [10, 11]. This is because, the method uses a number of concurrent `process`es to simulate task behaviours and OSEK/VDX OS scheduling behaviours in the application model. In the verification stage, these concurrent `process`es will generate a large number of interleavings which often result in state space explosion [12].

In order to make existing model checkers more scalable in verifying OSEK/VDX applications, one of reasonable solutions is to simplify the application models. In this paper, based on our previous research [13], we present the development of a software tool named autoC to address this problem. The tool automatically translates OSEK/VDX applications into equivalent sequential models which enable developers to verify an OSEK/VDX application by only verifying a sequential model (one concurrent `process`) rather than the application model that holds a lot of concurrent `process`es. Actually, there are several existing works in literatures that translate concurrent programs dispatched by a non-deterministic scheduler into sequential models. For example, Cimatti et al. [14] proposed an approach to translate SystemC concurrent programs into sequential programs. In addition, Ref. [15] also shows an approach to sequentialize the multi-threaded software that conforms to the POSIX standard [16]. However, these existing methods focus on a non-deterministic scheduler, and are not suitable to sequentialize OSEK/VDX applications.

Currently, our autoC accepts the C programming language as input, supports the 12 types of APIs specified in the OSEK/VDX OS standard, and can output two types of sequential model: Promela and C sequential models. The advantage of autoC is, based on the sequential translation, the multi-task OSEK/VDX application is translated into a sequential model. Thus, model checkers only verify one concurrent `process` instead of many concurrent `process`es in the verification stage. The two contributions of autoC are as follows: (i) the sequential translation of autoC enables developers to directly apply

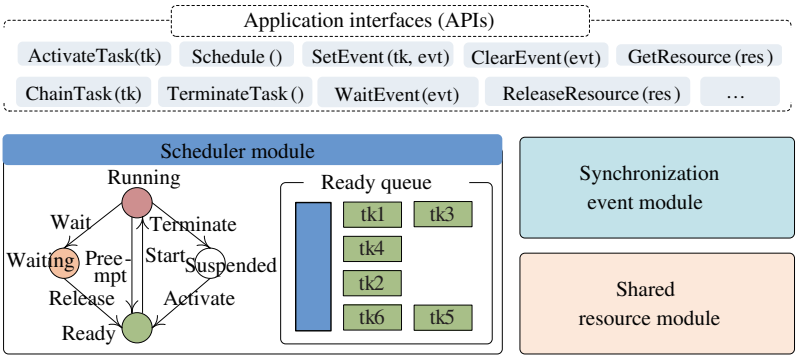

Figure 2 (Color online) Structure of OSEK/VDX OS and provided APIs.

different model checking techniques to verify OSEK/VDX applications according to different checking targets without considering the details of OSEK/VDX OS; (ii) autoC can be considered as a guideline for verifying other deterministic scheduler based multi-task software using model checking.

We conducted a series of experiments to evaluate the efficiency of autoC. In the conducted experiments, we firstly used autoC to translate the experimental applications into the equivalent sequential models, and then employed the well-known Spin model checker to carry out verification. Furthermore, in the experiments, an existing plain checking method [10, 11] was considered as a comparison object. Based on the experimental results, we found that autoC is not capable of efficiently translating OSEK/VDX applications into sequential models, but also of improving the efficiency and scalability of the Spin model checker in verifying OSEK/VDX applications compared with the plain checking method.

The outline of this paper is as follows. The OSEK/VDX OS and a running application used in the paper are introduced in Section 2. The plain checking method is presented in Section 3. In Section 4, the details of autoC are stated based on the running application. In addition, the advantages of autoC in contrast with the plain checking method are also described in this Section. The efficiency of autoC is demonstrated in Section 5, where the experiments and evaluation are discussed. We then compare our work with related work in Section 6. The last section concludes the paper.

2 Background of OSEK/VDX OS and running application

The OSEK/VDX supports the development of a customized application by providing multi-task development and many application interfaces (APIs) for developers. Details of the OSEK/VDX OS and a running application are stated below.

2.1 OSEK/VDX OS

In general, as shown in Figure 2, an OSEK/VDX OS consists of three primary process modules: a scheduler module, synchronization event module, and shared resource module. In addition, these process modules also provide many useful APIs to allow applications to change the scheduling order of tasks, to realize synchronous executions, and to access shared resources. The process modules of the OSEK/VDX OS and corresponding APIs are as follows.

2.1.1 *Scheduler module*

The OSEK/VDX OS allows developers to define two types of tasks in application: basic task and extended task. A basic task holds three states: running state, suspended state, and ready state. Compared with the basic task, an extended task can take synchronization events, and holds a unique state called waiting state. In the scheduling process, a deterministic scheduling policy, which is a static priority scheduling policy with mix-preemptive strategy (Full-preemptive strategy and Non-preemptive strategy), is adopted by the OSEK/VDX OS to conduct the executions of tasks, where a ready queue is used to manage the

Source file

```
int speed, fixspeed;
Task contask() {
    fixspeed=120;
    bool switch=true;
    while(switch=true){
        speed=read_speed();
        if(speed<fixspeed)
            ActivateTask(plustask);
        else
            ActivateTask(minustask);
        switch=read_switch();
    }
    TerminateTask();
}

Task plustask() {
    speed++;
    TerminateTask();
}

Task minustask() {
    speed--;
    TerminateTask();
}
```

Configuration file

```
Task contask
{
    Type=Basic;
    Priority=2;
    Schedule=Full;
    Autostart=True;
}

Task plustask
{
    Type=Basic;
    Priority=3;
    Schedule=Non;
    Autostart=False;
}

Task minustask
{
    Type=Basic;
    Priority=3;
    Schedule=Non;
    Autostart=False;
}
```

Figure 3 (Color online) Running application.

scheduling order of tasks. Furthermore, the OSEK/VDX scheduler provides several APIs for applications (e.g., TerminateTask, ActivateTask, Schedule and ChainTask), and tasks within an application can invoke these APIs to dynamically change the states of tasks. For example, when a task that is currently running invokes API ActivateTask(tk1) and the activated task tk1 is in the suspended state, task tk1 will be moved from suspended state to ready state by the OSEK/VDX scheduler.

2.1.2 *Synchronization event module*

The OSEK/VDX OS also supports a synchronization mechanism. The synchronization event module provides several APIs (e.g., SetEvent, WaitEvent and ClearEvent), and tasks within an application can invoke these APIs to implement the synchronous executions. For example, a running task t1 invokes the API WaitEvent(evt1), task t1 is waited until the event evt1 occurs by being set in another task using the API SetEvent(t1, evt1).

2.1.3 *Shared resource module*

The OSEK/VDX OS adopts the Priority Ceiling Protocol [17] to coordinate the task behaviours for accessing shared resources. The shared resource module provides two APIs for applications, i.e., GetResource and ReleaseResource. For example, if a task within an application needs to access a shared resource res1, the APIs GetResource(res1) and ReleaseResource(res1) can be invoked by the task to create a critical section for accessing the shared resource.

2.2 Running application

As shown in Figure 3, an OSEK/VDX application consists of two files: a source file, and a configuration file. The source file is used to present the concrete behaviors of an application, which can be developed in C programming language. The configuration file is used to configure an application, e.g., define tasks, synchronization events and shared resources. In task configuration, the attribute `Type` is used to indicate the type of task (basic or extended). The attribute `Priority` is used to set the priority of the task. `Schedule` is used to indicate the scheduling type of the task. If the attribute `Schedule` is set to `Full`, the task can be preempted by higher priority tasks; otherwise, this preemption is not possible. `Autostart` is used to specify the initial state of the task. If the attribute is set to `True`, the task starts from ready state as the initial state (it will be inserted into the ready queue according to the priority of the task); otherwise, the task starts from suspended state.

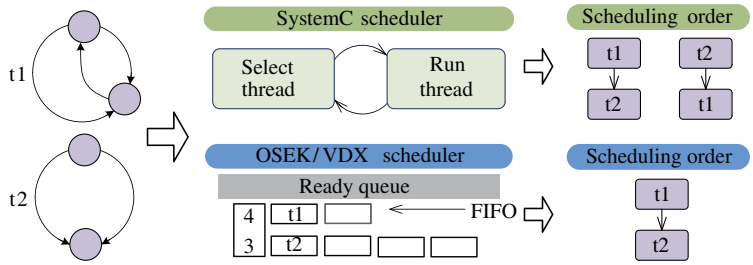

Figure 4 (Color online) Scheduling type: non-deterministic scheduler and deterministic scheduler.

The execution characteristics of OSEK/VDX applications are expressly explained by describing the symbolic execution of the running application shown in Figure 3 in this paragraph. As seen in Figure 3, there are three tasks, contask, plustask and minustask in the running application. When the application starts, since the only task in the ready state is contask (the attribute `Autostart` of contask is configured as `True`), the scheduler within the OSEK/VDX OS moves contask from ready state to running state. When contask is running on the processor, either the API ActivateTask(plustask) or ActivateTask(minustask) are invoked in the `if-else` branches. For instance, if ActivateTask(plustask) is invoked, the scheduler would be loaded to respond to this API, and then task plustask would be activated (scheduler moves plustask from suspended state to ready state). At this moment, the currently running task contask will be preempted by the activated task plustask, because the priority of contask is lower than that of plustask and its attribute `Schedule` is set to `Full`. Then, plustask will get processor to run and go to suspended state when the API TerminateTask() is invoked (API TerminateTask() is used to terminate a task, and the terminated task will be moved from running state to suspended state by the scheduler). When plustask is terminated, the scheduler dispatches contask to run again from the preempted point. Similarly, if ActivateTask(minustask) is invoked by contask, minustask will preempt contask to run and terminate itself when the API TerminateTask() is invoked.

Execution characteristics. According to the executions of running application, we can find the following execution characteristics of OSEK/VDX applications:

- Tasks within an application are concurrently executed based on the scheduling of the OSEK/VDX OS, and the running task can be explicitly determined by the OSEK/VDX scheduler.
- Tasks can invoke APIs to dynamically change the states of tasks, and these changed task states affects the scheduling order of tasks.

Due to the concurrency of tasks and dynamic scheduling, how to exhaustively check the developed OSEK/VDX applications is becoming a challenge for developers. In the following sections, we will demonstrate two exhaustive checking methods based on existing model checkers. The first method directly employs existing model checkers (we name this method as the plain checking method), and the other is based on our developed translator autoC.

3 Plain checking method

In general, in multi-threaded software such as in SystemC and ANSI-C concurrent programs, threads are concurrently executed, and the running thread, which is arbitrarily selected by scheduler, cannot be explicitly specified. For example, as depicted in Figure 4, if threads t1 and t2 are currently in the runnable or ready state and the SystemC scheduler is used to dispatch these two threads, there exist two possible scheduling orders, one of which is (t1, t2), and the other is (t2, t1). Usually, we refer to this type of scheduler as a non-deterministic scheduler. Due to the non-deterministic scheduling behaviours, it is difficult to achieve an exhaustive examination. In the software industry, this problem has been solved by applying model checking as an exhaustive technique. Currently, based on the different model checking techniques, many model checkers such as Spin, ESBMC and VCC have already been built for automatically verifying non-deterministic scheduler based concurrent programs. When we employ

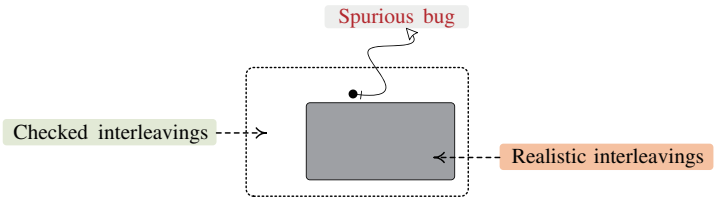

Figure 5 (Color online) Realistic task interleavings in deterministic scheduler based concurrent program.

existing model checkers to verify a non-deterministic scheduler based concurrent program, we usually do not need to consider the complex non-deterministic scheduling behaviours. In the verification stage, model checkers can systematically verify all possible scheduling orders or interleavings of threads. For example, if the Spin model checker is employed to verify the threads t1 and t2 depicted in Figure 4, we can simply use `proctype process` to represent each thread in the program model. In the verification stage, Spin verifies all of the possible scheduling orders of the threads.

Unfortunately, in contrast with the non-deterministic scheduler based multi-threaded programs, in OSEK/VDX applications, the running task can be explicitly determined by the OSEK/VDX scheduler based on the ready queue. For example, as depicted in Figure 4, we assume that tasks t1 and t2 are currently in the ready queue and the priority of task t1 is higher than that of task t2. If the OSEK/VDX scheduler is used to dispatch these two tasks, only a single scheduling order (t1, t2) exists. In general, we refer to this type of scheduler as deterministic scheduler. To verify this type of concurrent programs using existing model checkers, if we do not consider the scheduling behaviours and just use concurrent `process` to represent each task (like checking non-deterministic scheduler based multi-threaded programs), a lot of unnecessary interleavings of tasks will be checked by model checkers in the verification stage (as shown in Figure 5, the interleavings checked by model checkers are larger than the realistic interleavings in deterministic scheduler based concurrent program). Furthermore, due to the unnecessary interleavings, model checkers often find a spurious bug which makes the verification inexplicit. In order to explicitly verify OSEK/VDX applications using existing model checkers, we will demonstrate a checking method named the plain checking method based on our previous work [10]. In addition, the advantages and disadvantages of the plain checking method will also be discussed.

3.1 Key idea of plain checking method

Existing model checkers such as Spin can be enabled to explicitly verify OSEK/VDX applications by using the plain checking method, the key idea of which is: each of the tasks in the application, such as those that are used to check non-deterministic scheduler based concurrent programs, is designed as a concurrent `process`. In particular, a special concurrent `process` for OSEK/VDX OS is added in the application model to determine the running task and respond to the APIs invoked by tasks. Based on the application model with the OSEK/VDX OS `process`, all of the possible interleavings of tasks are checked by model checker; especially, unnecessary interleavings of tasks are removed by scheduling of the OSEK/VDX OS `process` in the verification stage.

According to the aforementioned idea, the plain checking method can explicitly verify OSEK/VDX applications using existing model checkers. For example, if we employ the Spin model checker to verify the running application, as shown in Figure 6, we can construct three `proctypes` to simulate contask, plustask and minustask; moreover, we can add a `proctype` named osekOS to represent the OSEK/VDX OS.

3.2 Advantage and disadvantage of plain checking method

In the plain checking method, since a special concurrent `process` for the OSEK/VDX OS is used to

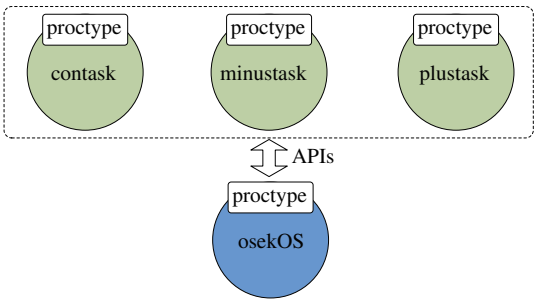

Figure 6 (Color online) Running application model in the Spin model checker.

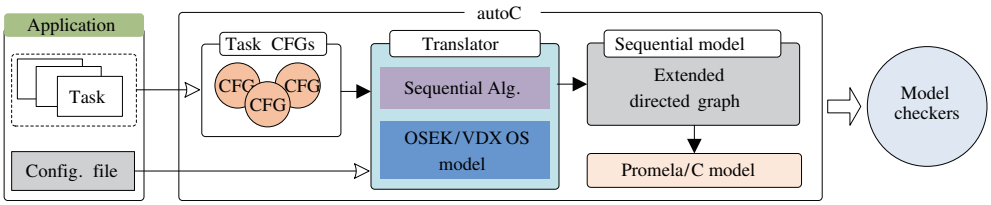

Figure 7 (Color online) Architecture of autoC.

explicitly dispatch tasks within an application and respond to the invoked APIs, the advantage of the plain checking method is that it can exhaustively verify OSEK/VDX applications without spurious bugs. However, the scalability of the method is limited, because the states of the OSEK/VDX OS `process` are explored by model checkers in the verification. If an application invokes a number of APIs, a large number of states from the OSEK/VDX OS `process` will be explored. This enables the model checkers to easily meet the verification limitation. Furthermore, in the plain checking method, the size of the application model (the number of concurrent `processes`) depends on the number of tasks. If a target application holds many tasks, in the verification stage, model checkers have to verify a large number of interleavings from task `process`es, and this could easily result in the state space explosion problem.

4 autoC

Because the application model consists of an excessive number of concurrent `proces`es, the plain checking method causes model checkers to fail in verifying a large-scale OSEK/VDX application which contains many tasks. Thus, a reasonable approach for improving the scalability of existing model checkers to verify OSEK/VDX applications would be to simplify the application models, e.g., to reduce the number of concurrent `process`es in the application model. In this section, we describe a software tool named autoC to address this problem by automatically translating an OSEK/VDX application into an equivalent sequential model. According to this model, in the verification stage, model checkers simply verify one concurrent `process` instead of multiple `process`es. The details of autoC are stated below.

4.1 Architecture of autoC

As shown in Figure 7, if autoC receives an OSEK/VDX application from users, it firstly calls the C Intermediate Language (CIL) [18] to interpret the behaviours of tasks written in complex C programming language into a simple `goto` program in the first step. Then, based on the simple `goto` program, the behaviours of tasks are extracted and constructed as corresponding control flow graphs (CFGs). In the second step, based on the task CFGs and configuration file, autoC translates the target application into an equivalent sequential model. Here, since the OSEK/VDX scheduler is used to dispatch tasks and tasks can invoke APIs to dynamically change the scheduling order, in order to translate an application into a sequential model equivalently, we embed an OSEK/VDX OS model in autoC to realize the scheduling behaviours and respond to the invoked APIs. In addition, an extended directed graph is used to compute

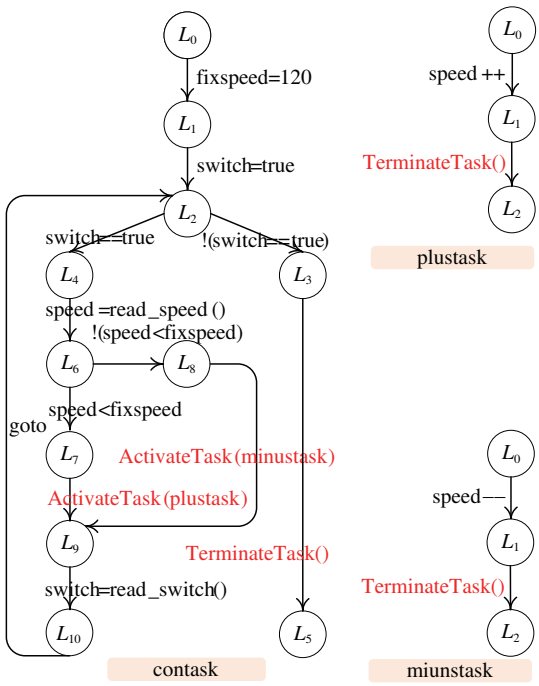

Figure 8 (Color online) CFGs for tasks contask, plustask and minustask.

the sequential model according to the executions of the target application. In the last step, autoC encodes the computed extended directed graph into the two types of sequential model, the Promela and C models.

4.2 Key processes in autoC

Based on the architecture of autoC, we systematically explain how autoC automatically translates OSEK/VDX applications into the equivalent sequential models.

4.2.1 *Task CFGs*

Developers can implement an OSEK/VDX application using C programming language. However, C code is often too complex to analyze in the static analysis. To easily analyze an OSEK/VDX application developed in C programming language, like the model checker CBMC [19], the application is firstly interpreted into the simple `goto` program based on CIL, where complex structures such as `loops` and `structs` in C code are interpreted as branch statements with `goto` labels and general variables. Then, the behaviours of all tasks within the application are extracted and constructed as the corresponding CFGs. The description of a task CFG has been represented in Definition 1. For example, as shown in Figure 8, autoC constructs three CFGs to represent each task included in the running application.

Definition 1. The CFG of a task is a tuple Ω=($\mathbb{L}$, L_0, Σ, R). Where, $\mathbb{L}=\{L_0, L_1, \ldots\}$ is a set of statement locations of a task with the start location $L_0 \in \mathbb{L}$. Σ is a set of task statements, the expression of a statement $\alpha \in \Sigma$ is as follows:

$$\alpha ::= \text{condition} \,|\, \text{assignment} \,|\, \text{goto} \,|\, \text{assertion} \,|\, \texttt{API},$$

$R \subseteq \mathbb{L} \times \mathbb{L}$ is the set of directed edges labelled by task statements Σ.

4.2.2 *Translation*

To translate an OSEK/VDX application into the equivalent sequential model, there are two problems that should be addressed, one is how to explicitly perform the scheduling behaviours of OSEK/VDX OS, and the other is how to compute the sequential model.

As to explicitly perform the scheduling behaviours of OSEK/VDX OS, we embed an OSEK/VDX OS model in autoC for dispatching tasks and responding to the invoked APIs. The embedded OSEK/VDX OS model consists of two key components (see [13]). The first component is the data structures used to record the scheduling data such as the states of the tasks. The second component is the functions that are used to respond to APIs and update the data within the data structures. In addition, in order to successfully compute the sequential model of an OSEK/VDX application, the extended directed graph defined in Definition 2 is employed to carry out sequential translation. In the translation processes, autoC uses the extended directed graph to execute an application in a symbolic way, and calls the embedded OS model to determine the running task when meeting an API.

Definition 2. An extended directed graph G is a tuple $G=(V, v_0, E)$. Where, V is a set of nodes of G with the start node $v_0 \in V$. A node $v \in V$ consists of two variables p and D, p is an array used to record the statement locations of tasks, and D is a set of variables used to store the states of tasks, the states of synchronization events and the states of shared resources. $E \subseteq V \times V$ is a set of directed edges used to map the statements of tasks within an application.

Based on the extended directed graph and embedded OSEK/VDX OS model, the key processes of autoC for translating a given OSEK/VDX application into a sequential model are formalized in Algorithm 1. In the translation processes, autoC does not compute the values of variables; instead, it just explores task statements according to the executions of running task. If the explored statements are sequential statements and branch statements, autoC will create several new nodes in the extended directed graph,

Algorithm 1 The key processes of autoC

Input: CFGs of takes and configuration file of application
Output: Extended directed graph G
Create a start node v_0 and a set V for extended directed graph G;
Initialize p and D in node v_0 with initial locations of task CFGs and application configuration file, $V := \{v_0\}$;
Call OS model to determine running task t in node v_0 based on D of node v_0;
Create a node v and a set V', $v := v_0$, $V' := \{v\}$;
while $V' \neq \emptyset$ **do**
  $v \in V'$, $V' := V' \backslash \{v\}$;
  if running task t in node v is not null **then**
    explore task statements from the current location of running task t in node v in symbolic way;
    if explored statement is an API **then**
      create a new node v', where $v' := v$;
      map the explored statement in directed edge (v, v');
      update p of node v' with current location of running task t in node v;
      call OS model to respond to the API and determine running task t in node v' based on D of node v';
      $V' := V' \cup \{v'\}$;
    end
    if explored statements are branch statements **then**
      according to the number m of branches, create m new nodes $v'_1,...,v'_m$, where $v'_1 := v,..., v'_m := v$;
      map the explored branch statements in directed edges $(v, v'_1),...,(v, v'_m)$, respectively;
      update p of node $v'_1,...,v'_m$ with current location of running task t in node v;
      $V' := V' \cup \{v'_1,...,v'_m\}$;
    else
      create a new node v', where $v' := v$;
      map the explored statement in directed edge (v, v');
      update p of node v' with current location of running task t in node v;
      $V' := V' \cup \{v'\}$;
    end
    forall $v_i \in V'$ **do**
      if $v_i = v_j \in V$, then change the relationship of directed edge (v, v_i) to (v, v_j), $V' := V' \backslash \{v_i\}$; otherwise,
      $V := V \cup \{v_i\}$ (where $v_i = v_j$ means that p and D in node v_i are equal to p and D in node v_j, respectively);
    end
  end
end
return G;

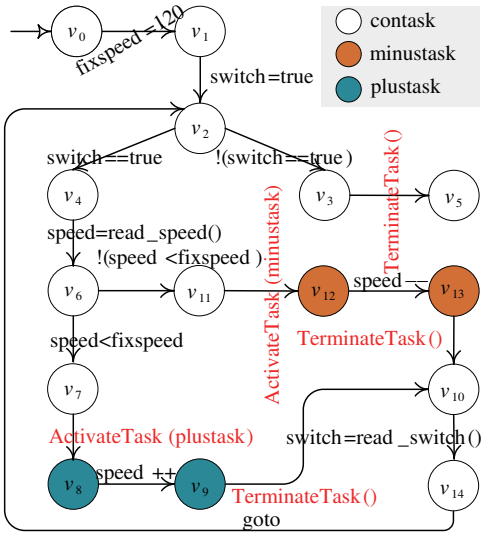

Figure 9 (Color online) Extended directed graph for running application.

and then maps the explored statements in the directed edges. When the explored statement is an API, autoC firstly maps the statement in the extended directed graph and then calls the embedded OS model to respond to the API and determine the next running task. Particularly, if the task locations p and the scheduling data D in a new node are equal to those of an old node, autoC constructs a cycle in the extended directed graph.

4.2.3 *Sequential model*

Once the extended directed graph is constructed according to the executions of the given application, autoC encodes the extended directed graph into the Promela and C sequential models to enable developers to employ existing model checkers to carry out the verification directly. Actually, based on the constructed extended directed graph, developers can easily select different model checkers to verify a given OSEK/VDX application according to the different checking targets. This becomes possible because task statements and the relationships between them are clearly shown in the extended directed graph. Developers only need to implement an encoder to encode the extended directed graph into the input languages of selected model checkers.

4.3 Example

We include this example to facilitate the understanding of the translation or sequentialization processes of autoC. This example is based on the running application depicted in Figure 3. As shown in Figure 9, in the first step, autoC creates a start node v_0 and initializes v_0 with the initial locations of tasks and configuration file (in node v_0, all tasks start from initial locations L_0 shown in Figure 8 with contask in the ready state, plustask and minustask in the suspended state). In the second step, autoC calls the embedded OS model to determine the running task in node v_0. Currently, contask becomes the running task in node v_0. Then, autoC successively explores the statements of contask and maps the explored statements in the directed edges of G. For example, in Figure 9, two nodes v_1 and v_2 are created, and the explored statements are mapped to directed edges (v_0, v_1) and (v_1, v_2). When autoC explores a statement from node v_2, it meets the condition statements of `while` loop in contask. Thus, the two nodes v_3 and v_4 are created, and the loop condition statement and negative condition statement are mapped to directed edges (v_2, v_4) and (v_2, v_3). The task statements from node v_3 are explored by invoking the API TerminateTask. Then, autoC creates a node v_5 and maps the API in directed edge (v_3, v_5). Since the explored statement is an API, the OS model is called to respond to the API and compute the states of tasks (in node v_5, all tasks are in the suspended state). When autoC explores the task statements

```
int speed, fixspeed;
bool switch; //local variable in contask
void seqM() {
    fixspeed=120;
    switch=true;
Loop:
    if(switch== true) {
        speed=read_speed ();
        if(speed<fixspeed) {
            //ActivateTask(plustask );
            speed ++;
            //TerminateTask ();
        }
        else if(!(speed  < fixspeed)) {
            //ActivateTask(minustask );
            speed—;
            //TerminateTask ();
        }
        switch=read_switch();
    }
    goto Loop;
}
```

Figure 10 (Color online) C sequential model for running application.

from node v_4, according to the executions of contask, autoC creates five nodes $(v_6, v_7, v_8, v_{11}, v_{12})$ to map the corresponding task statements, e.g., the APIs ActivateTask(plustask) and ActivateTask(minustask) are mapped in directed edges $(v_7,\ v_8)$ and $(v_{11},\ v_{12})$, respectively. In edge $(v_7,\ v_8)$, since the API ActivateTask(plustask) is invoked, autoC calls the OS model to respond to the API and compute the states of tasks in node v_8. At this moment, contask is preempted by plustask in node v_8, and then autoC creates two nodes v_9 and v_{10} to successively map the statements of plustask. Since the mapped statement in directed edge $(v_9,\ v_{10})$ is the API TerminateTask, the OS model is called to respond to the API and to compute the states of tasks in node v_{10} (in node v_{10}, contask becomes running task again). Accordingly, autoC also maps the task statements of minustask in directed edges $(v_{12},\ v_{13})$ and $(v_{13},\ v_{10})$ from node v_{12}. After that, autoC creates a node to continue the executions of contask from node v_{10} and maps the corresponding statements in directed edges.

Then, autoC encodes the computed extended directed graph into the sequential model, e.g., if we require autoC to generate a C model, as shown in Figure 10, autoC will encode the extended directed graph depicted in Figure 9 into the corresponding C sequential model. Based on the shown C sequential model, we can easily find that the C sequential model is equivalent to the executions of the running application. In particular, since OSEK/VDX OS model is embedded in autoC to dispatch tasks and respond to the invoked APIs, the sequential model does not hold the states of OSEK/VDX OS.

4.4 Advantages of autoC

One of the main advantages of autoC is that the translation or sequentialization process of autoC obviates the need for developers to consider the complex behaviours of the OSEK/VDX OS when using model checkers based on different model checking techniques to verify OSEK/VDX applications, such as the symbolic model checking [20], bounded model checking [21], and counterexample guided abstraction refinement [22, 23] techniques. This enhances the cost-effectiveness of the verification of OSEK/VDX applications. In addition, based on the sequential model translated by autoC, advanced software techniques for sequential programs (e.g., testing [24] technique and code optimization [25] technique) can be directly employed to process OSEK/VDX applications.

4.5 Usage of autoC

autoC is available on the website http://www.jaist.ac.jp/~s1220209/autoC.htm, implemented on the Visual Studio 2010 platform with C++ programming language. Currently, it can be run on the Windows OS, and accepts C programming language as input. In addition, autoC supports 12 types of APIs specified in the OSEK/VDX standard, and it can output two types of sequential models, the Promela and C sequential models for model checking OSEK/VDX applications. Furthermore, to enable developers to

easily verify OSEK/VDX applications using different model checkers, autoC can also output a sequential model represented by a directed graph. Based on the relationships between nodes and labelled statements in the directed graph model, developers can easily encode it into the input languages of selected model checkers.

The use of autoC firstly requires Visual Studio to be installed, after which autoC can be run from the Visual Studio command prompt. For example, if we require autoC to translate the running application shown in Figure 3 into sequential model and output C model, we can run autoC on the Visual Studio command prompt with the command "`autoC -sourcefile` source.`cpp -configfile` config.`oil -seqC`." Further details of autoC are provided on the autoC website.

5 Experiments and discussion

5.1 Experiments

we have conducted a series of experiments to evaluate the efficiency of autoC. The sequential translation features of autoC determine the extent to which the task number and API number affect the performance of autoC. Thus, autoC was comprehensively investigated on realistic applications by selecting experimental applications with different task and API numbers as benchmarks. Moreover, the execution behaviours of OSEK/VDX applications were realistically represented by additionally taking into account the non-preemptive scheduling behaviour, full-preemptive scheduling behaviour, mix-preemptive scheduling behaviour, synchronous behaviour, and accessing shared resource behaviour of the selected benchmarks. The benchmarks developed by our research group are available at http://www.jaist.ac.jp/~s1220209/autoC.htm. In addition, the well-known Spin model checker was selected to show whether autoC can improve the scalability of existing model checkers for verifying OSEK/VDX applications. Particularly, in the experiments, the plain checking method based on the Spin model checker was considered as comparison object.

All the experiments were conducted on the Intel Core(TM)i7-3770 CPU with 32 G RAM, and we set the time limit and memory limit to 600 s and 1 GB, respectively. Note that, in the experiments, the "`C compiler`" of Spin was set to "−DVECTORSZ=16384 −DBITSTATE", and the maximum depth was set to "20000000". In addition, we thoroughly evaluated autoC by not specifying any property in the experiments to allow Spin to explore the states of the target benchmarks as much as possible. The experimental results are listed in Table 1. In the table, `#t` is the number of tasks, `#API` is the times of invoked APIs by tasks, `#s` is the number of explored states by Spin. "Mb" and "Time" are the memory consumption and time consumption measured in Mbyte and seconds, respectively. M.O. and T.O. stand for that Spin model checker runs out of time and memory, respectively.

5.2 Discussion

The experimental results shown in Table 1 indicate that the plain checking method fails to verify the benchmarks which contain a number of tasks and APIs (e.g., lines 4, 12 and 20). This is because, (i) in the plain checking method, the application model holds too many concurrent `process`es, thereby easily causing the state space to experience exponential growth in the verification stage. In addition, (ii) in the method, when an API is invoked by the running task, the states of OSEK/VDX OS are checked, and this significantly increases the state space if an application holds a number of APIs. These drawbacks seriously limit the efficiency and scalability of this method.

In contrast with the plain checking method, the autoC+Spin method can successfully verify these benchmarks with lower costs (time and memory) and lesser states. This is because, (i) based on the sequentialization of autoC, the model checker Spin only verifies one concurrent `process` rather than multiple concurrent `process`es. In addition, (ii) in the sequentialization, we embedded an OSEK/VDX OS model in autoC to explicitly dispatch tasks and respond to the invoked APIs, thereby resulting in

Table 1 Comparison: Spin based Plain checking method vs. autoC+Spin

Benchmark	#t	#API	autoC + Spin							
			Spin$^{\texttt{plain checking method}}$			autoC		Spin		
			#s	Time	Mb	Time	Mb	#s	Time	Mb
Non-preemption	6	11	4652	0.21	17.7	0.18	3.19	36	0	0.02
	8	14	25253	0.71	81.1	0.26	3.64	38	0.01	0.04
	10	17	103835	2.67	303.2	0.30	4.11	60	0.01	0.09
	18	35	–	T.O.	–	0.60	5.72	99	0.02	0.50
Full-preemption	4	61	20501	0.10	82.5	0.14	2.98	1008	0.11	0.93
	6	101	34371	1.12	126.2	0.19	3.34	1624	0.16	1.79
	9	161	46990	1.26	138.1	0.29	4.05	2607	0.27	2.78
	13	241	–	–	M.O.	0.45	5.14	3904	0.39	0.75
Mix-preemption	5	101	13541	0.44	54.7	0.18	3.21	1308	0.13	0.75
	9	161	–	–	M.O.	0.30	4.22	2588	0.13	1.71
	13	241	–	–	M.O.	0.45	5.23	3868	0.26	2.56
	13	313	–	–	M.O.	0.41	4.78	390	0.38	0.24
Synchronization	5	14	2443	0.15	11.6	0.18	3.16	134	0.04	0.11
	8	23	24159	0.78	86.8	0.26	4.01	230	0.02	0.14
	11	32	210841	5.37	612.4	0.42	4.53	322	0.02	0.02
	12	42	382329	9.62	998.3	0.45	5.11	358	0.03	0.19
Shared resource	2	4	13907	0.46	56.5	0.21	3.41	1308	0.13	0.75
	9	320	–	–	M.O.	0.34	4.63	2588	0.26	1.71
	13	480	–	–	M.O.	0.61	5.84	3868	0.38	2.56
	12	250	–	–	M.O.	0.44	4.98	389	0.03	0.24

the sequential model only holding the states of the given application. These efforts improve the efficiency and scalability of the Spin model checker in verifying large-scale OSEK/VDX applications.

Based on the results of comparison, it can be seen that, autoC is not only capable of efficiently translating OSEK/VDX applications into sequential models, but also of improving the efficiency and scalability of existing model checkers for verifying OSEK/VDX applications.

6 Related work

Currently, the OSEK/VDX standard has been widely adopted by many automotive manufacturers and research groups to implement practical systems and study embedded systems, e.g., well-known manufacturers BMW and TOYOTA, and research groups IRCCyN and TOPPERS. However, ensuring the reliability of the developed OSEK/VDX OS and its applications has become a challenge for developers with the continuously increasing complexity of the development process.

In the scope of checking developed OSEK/VDX OS, there are some invaluable methods, e.g., Chen and Aoki [26] have proposed a method to generate the highly reliable test-cases for checking whether the developed OS conforms to the OSEK/VDX OS standard based on Spin. In addition, for Trampoline[1)], which is an open source RTOS that was developed based on the OSEK/VDX standard, Choi [27] presented a method to convert the Trampoline kernel into formal models, and an incremental verification approach is proposed to carry out the verification in the method. Furthermore, a CSP-based approach for checking the code-level OSEK/VDX OS is presented in [28]. All of these related studies are different from our work, because our autoC focuses on developed OSEK/VDX applications.

For the developed OSEK/VDX applications, we have proposed a Spin-based checking method to check the safety property [10]. Similarly, the paper [11] also proposed a method to check the timing property based on the model checker UPPAAL. Although these two methods can explicitly verify OSEK/VDX applications, they cannot handle the large-scale applications that hold many tasks because of a lot of

1) Trampoline. http://trampoline.rts-software.org/.

concurrent `processes` of tasks and too many details of OSEK/VDX OS included in the application model. To avoid the states of OSEK/VDX OS to be checked in the verification, we presented a new technique named EPG [29] based on the SMT-based bounded model checking. Even so, the method is not efficient to check the applications which holds a lot of branches, since the method will spend much time exploring all of the execution paths in order to construct the corresponding transition system, which will seriously slow down the performance of the method. Compared with EPG technique, autoC uses an extended directed graph to execute OSEK/VDX applications in symbolic way rather than exploring execution paths, which is more efficient than EPG technique.

To verify concurrent programs with a scheduler, Liu and Joseph [30] proposed a transformational approach to specification and verification of concurrent application programs executing on systems with limited resources. There, a generic transformation is defined such that for an application (or a specification), a system model with limited resources such as the number of processors and their computational speeds, and a scheduler (or a specification of the scheduling policy), the implementation of the application on the system by the scheduler is transformed into a program with reduced interleavings. The transformation is used for the verification of the schedulability of real-time and fault-tolerant applications, and the feasibilities of real-time schedulers for the real-time and fault-tolerance requirements of applications. The advantage is to allow the real-time and fault-tolerance requirements be specified and verified in the traditional theory refinement and temporal verification in a single notation. However, that work does not consider the issue of a possible implementation of the transformation for any existing operating system, and to our best knowledge there does not exist any tool to support Liu and Joseph's transformational approach. In this paper, we do not consider the verification process, but we focus on the problems of design and implementation of a transformation for translating OSEK/VDX concurrent applications into sequential execution models. In the field of verifying concurrent programs using sequentialization based model checking, several studies, such as [14,15], have been carried out. In these studies, the target systems are non-deterministic scheduler based concurrent programs such as SystemC and ANSI-C. In the translation process, random numbers are appended in the target programs to simulate the non-deterministic scheduling behaviour. However, as reported in [14], sequentialization based model checking methods are usually unable to verify large-scale programs, because too many appended random numbers often incur a large number of non-deterministic states, which will quickly cause the model checkers to reach their verification limit. Compared with existing works, autoC focuses on a deterministic scheduler. This involves employing an extended directed graph for sequential translation to compute a sequential model that does not hold random numbers. Once an OSEK/VDX application is successfully translated into a sequential program by autoC, model checkers can efficiently verify the target application.

7 Conclusion

This paper presented a software tool named autoC that is capable of automatically translating the deterministic scheduler based OSEK/VDX applications into equivalent sequential models. We explicitly realized the scheduling behaviour of the OSEK/VDX OS by embedding an OSEK/VDX OS model in autoC to dispatch tasks within applications and respond to the APIs invoked from tasks. In addition, an extended directed graph was used to carry out sequential translation in order to efficiently sequentialize OSEK/VDX applications. The sequential models translated by autoC enable developers to directly employ advanced model checking techniques to verify OSEK/VDX applications; moreover, developers can also use state-of-the-art software techniques for sequential programs to process OSEK/VDX applications. We conducted a series of experiments to evaluate the efficiency of autoC. The experimental results show that autoC not only has the ability to efficiently sequentialize OSEK/VDX applications, but also to improve the efficiency and scalability of existing model checkers in verifying large-scale OSEK/VDX applications.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61602224, 61472240) and Fundamental Research Funds for the Central Universities (Grant Nos. lzujbky-2016-142, lzujbky-2016-k07).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Lemieux J. Programming in the OSEK/VDX Environment. New York: CMP Media, Inc., 2011
2 Clarke E M, Emerson E A, Sifakis J. Model checking: algorithmic verification and debugging. Commun ACM, 2009, 152: 74–84
3 Clarke E M, Grumberg O, Long D E. Model checking and abstraction. ACM Trans Programm Languages Syst, 1994, 16: 1512–1542
4 Pu F, Zhang W H. Combining search space partition and abstraction for LTL model checking. Sci China Ser F-Inf Sci, 2007, 50: 793–810
5 Holzmann G J. The Spin Model Checker: Primer and Reference Manual. Boston: Lucent Technologies Inc., 2003
6 Morse J, Ramalho M, Cordeiro L, et al. ESBMC 1.22. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2014. 405–407
7 Cohen E, Dahlweid M, Hillebrand M, et al. VCC: a practical system for verifying concurrent C. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. Berlin: Springer, 2009. 23–42
8 Cordeiro L, Fischer B. Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proceedings of the 33rd International Conference on Software Engineering, Waikiki, 2011. 331–340
9 Cimatti A, Micheli A, Narasamdya I, et al. Verifying SystemC: a software model checking approach. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, Lugano, 2010. 51–60
10 Zhang H T, Aoki T, Chiba Y. A spin-based approach for checking OSEK/VDX applications. In: Proceedings of the 3rd International Workshop Formal Techniques for Safety-Critical Systems (FTSCS), Paris, 2014. 239–255
11 Waszniowski L, Hanzalek Z. Formal verification of multitasking applications based on timed automata model. J Real-Time Syst, 2008, 38: 39–65
12 Clarke E M, Klieber W, Novacek M, et al. Model checking and the state explosion problem. In: Tools for Practical Software Verification. Berlin: Springer, 2012. 1–30
13 Zhang H T, Aoki T, Chiba Y. Yes! You can use your model checker to verify OSEK/VDX applications. In: Proceedings of the 8th International Conference on Software Testing, Verification and Validation, Graz, 2015. 1–10
14 Campana D, Cimatti A, Narasamdya I, et al. An analytic evaluation of SystemC encodings in Promela. In: Proceedings of the 18th International SPIN Conference on Model Checking Software, Snowbird, 2011. 90–107
15 Inverso O, Tomasco E, Fischer B, et al. Lazy-CSeq: a lazy sequentialization tool for C. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Grenoble, 2014. 398–401
16 IEEE Standard for Information Technology. Portable operating system interface (POSIX) base specifications. Issue 7. http://standards.ieee.org/reading/ieee/interp/1003.1-2008.html
17 Burns A, Wellings A. Real-Time Systems and Programming Languages. 4th ed. New York: Addison Wesley Longmain, 2009
18 George C N, Scott M, Shree P, et al. CIL: intermediate language and tools for analysis and transformation of C programs. In: Proceedings of the 11th International Conference on Compiler Construction. London: Springer, 2002. 213–228
19 Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programings. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2004. 168–176
20 Yang Z J, Wang C, Gupta A, et al. Model checking sequential software programs via mixed symbolic analysis. ACM Trans Design Autom Electron Syst, 2009, 14: 1–26
21 Armin B, Clarke E M, Zhu Y S. Bounded model checking. Adv Comput, 2003, 58: 117–148
22 Tian C, Duan Z H. Detecting spurious counterexamples efficiently in abstract model checking. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 202–211
23 Tian C, Duan Z H, Duan Z. Making CEGAR more efficient in software model checking. IEEE Trans Softw Eng, 2014, 40: 1206–1223
24 Basili V R, Selby R W. Comparing the effectiveness of software testing strategies. IEEE Trans Softw Eng, 1987, 13: 1278–1296
25 Hoffmann J, Ussath M, Holz T, et al. Slicing droids: program slicing for smali code. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, Portugal, 2013. 1844–1851
26 Chen J, Aoki T. Conformance testing for OSEK/VDX operating system using model checking. In: Proceedings of the 18th Asia-Pacific Software Engineering Conference, Washington, 2011. 274–281
27 Choi Y J. Safety analysis of trampoline OS using model checking: an experience report. In: Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering, Washington, 2011. 200–209
28 Huang Y H, Zhao Y X, Zhu L F, et al. Modeling and verifying the code-level OSEK/VDX operating system with CSP. In: Proceedings of the 5th International Symposium on Theoretical Aspects of Software Engineering, Xi'an, 2011. 142–149
29 Zhang H T, Aoki T, Lin X H, et al. SMT-based bounded model checking for OSEK/VDX applications. In: Proceedings of the 20th Asia-Pacific Software Engineering Conference, Bangkok, 2013. 307–314
30 Liu Z M, Joseph M. Specification and verification of fault-tolerance, timing, and scheduling. ACM Trans Program Lang Syst, 1999, 21: 46–89