PAPER

Verifying OSEK/VDX Applications: A Sequentialization-Based Model Checking Approach*

Haitao ZHANG^{†a)}, *Nonmember*, Toshiaki AOKI[†], *Member*, and Yuki CHIBA[†], *Nonmember*

SUMMARY OSEK/VDX, a standard for an automobile OS, has been widely adopted by many manufacturers to design and develop a vehicle-mounted OS. With the increasing functionalities in vehicles, more and more complex applications are being developed based on the OSEK/VDX OS. However, how to ensure the reliability of developed applications is becoming a challenge for developers. To ensure the reliability of developed applications, model checking as an exhaustive technique can be applied to discover subtle errors in the development process. Many model checkers have been successfully applied to verify sequential software and general multi-threaded software. However, it is hard to directly use existing model checkers to precisely verify OSEK/VDX applications, since the execution characteristics of OSEK/VDX applications are different from the sequential software and general multi-threaded software. In this paper, we describe and develop an approach to translate OSEK/VDX applications into sequential programs in order to employ existing model checkers to precisely verify OSEK/VDX applications. The value of our approach is that it can be considered as a front-end translator for enabling existing model checkers to verify OSEK/VDX applications.

key words: OSEK/VDX applications, Formal Software Verification, deterministic scheduler, sequentialization

1. Introduction

OSEK/VDX [2], [3], a standard for an automobile OS, has been widely adopted by many automobile manufacturers to design and develop a vehicle-mounted OS, such as BMW, Opel and Volkswagen. With the increasing demands for driving fun and safety in vehicles, more and more complex applications are developed based on the OSEK/VDX OS. However, how to ensure the reliability of the developed OSEK/VDX applications is becoming a challenge for developers with the increasing complexity in development work.

To ensure the reliability of the developed OSEK/VDX applications, model checking [4], [5] as an exhaustive technique can be used to verify OSEK/VDX applications for discovering subtle errors in the development process. There exist many model checking methods that have been applied to verify the sequential software [6] and general multi-threaded software [7], [8] such as systemC programs [9] and ANSI-C programs. However, it is hard to directly use these existing model checking methods to precisely verify the OSEK/VDX applications, since the execution characteristics

of OSEK/VDX applications are different from the sequential software and general multi-threaded software. E.g., when an application runs on the OSEK/VDX OS, the executions of tasks within the application are dispatched by a scheduler, and the running task is explicitly determined by the scheduler according to the task priorities and configuration data. Moreover, tasks can invoke service APIs supported by OSEK/VDX OS to dynamically change the states of tasks, synchronization events and shared resources defined in the application, and the changed states will haphazardly affect the scheduling of tasks.

If the model checking methods for sequential software are employed to verify OSEK/VDX applications, we have to translate the target OSEK/VDX application into a sequential program in advance since it is more like multi-threaded software compared with the sequential software. Furthermore, if we directly apply the existing model checking methods for general multi-threaded software to verify OSEK/VDX applications, it is too imprecise because a lot of unnecessary interleavings of tasks will be checked by existing methods, especially these unnecessary interleavings will usually result in a spurious bug in the verification. This is because, in the existing works for the general multi-threaded software, since the running thread cannot be explicitly determined, all of the possible interleavings of runnable threads are taken into account in the verification in order to completely check the target software. However, in contrast with the general multi-threaded software, in OSEK/VDX applications the running task is explicitly determined by the OSEK/VDX scheduler. According to the different scheduling policy between the general multi-threaded software and OSEK/VDX applications, we can easily find that the existing model checking methods for the general multi-threaded software are also unsuitable to check the developed OSEK/VDX applications.

To precisely verify OSEK/VDX applications using model checking technique, in our previous work [10] we presented an approach based on the Spin model checker [11]. In the approach, a synchronization model (synM), which is a combination model of application model and OSEK/VDX OS model, is employed to simulate the executions of the application. We have implemented this approach and conducted many experiments. The experimental results show that the synM approach can precisely verify the safety property of the OSEK/VDX applications, but the capability of this approach is limited because too many details about OS model are checked in the verification stage, especially the state space explosion [12] will happen if the

Manuscript received February 5, 2015.

Manuscript revised June 2, 2015.

Manuscript publicized July 6, 2015.

[†]The authors are with the School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Nomi-shi, 923–1292 Japan.

*The preliminary version is presented in [1].

a) E-mail: zhanghaitao@jaist.ac.jp

DOI: 10.1587/transinf.2015EDP7043

checked application invokes a lot of service APIs.

To avoid the behaviours of OS model to be checked in the verification stage and intend to handle a complex OSEK/VDX application which holds a large number of states, in a later work [13], [14] we presented a new technique named execution path generator (EPG) to verify OSEK/VDX applications based on the SMT-based bounded model checking (BMC) [15], [16]. We have conducted many experiments using EPG technique. The experimental results show that EPG technique is an efficient method to verify the applications which hold a lot of tasks and service APIs with lower cost including states, time and memory than synM approach. However, it is not efficient to check the applications which include a lot of branches and loops with APIs. Moreover, as well known, it is difficult and impractical to implement a new model checker within a short time, since a basic model checker usually consists of several complex modules such as the language parsing, simulation, verification, counterexample generation and display. To finish all of these modules with reasonable soundness guarantee often takes years of effort, e.g., the established model checkers Spin [11] and UPPAAL [17] are the result of decades of development. Therefore, in order to reliably and efficiently verify the OSEK/VDX applications which hold a lot of tasks, loops and APIs, we propose an alternative approach to translate OSEK/VDX applications into sequential programs for enabling existing model checkers to verify OSEK/VDX applications in this paper.

In our approach, we use a directed graph to represent the sequential program of an OSEK/VDX application, and a directed graph constructor as a simulator is developed to construct the directed graph for the OSEK/VDX application. In addition, two optimization strategies in the scheduling level for cutting the unnecessary states and accelerating the speed of bug hunting are also proposed in this paper. Note that our approach is not limited to the OSEK/VDX applications but can also be applied in such multi-tasks/threads software in which tasks/threads can invoke service APIs to interact with its OS, and a deterministic scheduling policy (such as static priority scheduling policy) is adopted by scheduler to dispatch tasks/threads. To the best of our knowledge, there is no work that considers the translation from the software like OSEK/VDX applications to sequential programs. The main contribution of our paper is that it proposes an approach to sequentialize OSEK/VDX applications.

We have implemented a prototype tool named `autoC`[†] according to the proposed approach. Currently, the tool supports the C programming language without `pointer`, `struct` and function calls as input language. We also evaluated our approach based on a series of experiments. In the experiments, as to comprehensively investigate the effectiveness of our approach, the OSEK/VDX applications which hold different task number, API number and loop number are selected as our benchmarks. According to the conducted experiment results, we find, based on the sequen-

tialization process of our approach, the selected back-end model checker Spin can efficiently verify the given applications with the less cost in terms of states, time and memory compared with the synM method and EPG technique, especially when the state reduction and acceleration scheduling strategies are useful to cut the unnecessary states and accelerate the speed of bug hunting. Moreover, the experiment results indicate that our approach can be considered as a practical method to check the realistic OSEK/VDX applications, since the back-end model checker has a very small verification cost.

The rest of the paper is structured as follows. The preliminaries for the OSEK/VDX OS and a motivating example are presented in Sect. 2. Based on the discussion about the execution characteristics of the motivating example, the approach for translating OSEK/VDX applications into sequential programs is demonstrated in Sect. 3. As to show the effectiveness and practicability of our approach, the implementation and experiments are carried out in Sect. 4. Related work is discussed in Sect. 5. Conclusion and future work are placed in the last section.

2. Preliminaries

2.1 OSEK/VDX OS

A general OSEK/VDX OS consists of a scheduler module, synchronization event process module, shared resource process module, alarm process module and interruption process module. Based on these system modules, OSEK/VDX OS supports a standardized application interfaces (APIs) for user to develop customized applications. In our research, we focus on the applications that communicate with scheduler module, synchronization event process module and resource process module. The structure of OSEK/VDX OS with an application is shown in Fig. 1.

Scheduler module: OSEK/VDX OS can process two types of tasks, basic task and extended task. The states of a basic task consist of *running* state, *suspended* state, and *ready* state. Compared with the basic task, the extended task can hold synchronization events and has a unique state called *waiting* state. In the scheduling process, the *static priority scheduling policy* with non-preemptive and full-preemptive strategies is adopted by the scheduler to conduct the executions of tasks, and moreover, scheduler manages a ready queue to indicate the execution order of tasks. Besides, scheduler can respond to four service APIs (*TerminateTask*, *ActivateTask*, *ChainTask*, and *Schedule*) that can be invoked by tasks to switch task states. For instance, if the service API *ActivateTask*(*tk1*) is invoked by running task, and task *tk1* is currently in the *suspended* state, scheduler will move task *tk1* from *suspended* state to *ready* state.

Synchronization event process module: In the synchronization event process module, OSEK/VDX OS provides a synchronization mechanism for implementing synchronous executions between tasks. Particularly, only extended tasks can hold a definite number of events, and events are the cri-

[†]<http://www.jaist.ac.jp/~s1220209/autoC.htm>

Fig. 1 OSEK/VDX OS structure with an application

teria for the switching of task states from *running* state to *waiting* state or from *waiting* state to *ready* state. There are three service APIs (*SetEvent*, *WaitEvent*, and *ClearEvent*) that can be responded by event process module, and tasks can invoke these service APIs to implement the synchronous executions. E.g., when the running task *tk1* waits for the event *evt1* using service API *WaitEvent(evt1)*, task *tk1* cannot continue until the event *evt1* is set by other tasks (basic tasks or extended tasks) using service API *SetEvent(tk1,evt1)*.

Shared resource process module: OSEK/VDX OS adopts the *Priority Ceiling Protocol* [18] to coordinate the behaviors of task accessing shared resources in the resource process module. The resource process module supports two service APIs (*GetResource* and *ReleaseResource*) which can be invoked by tasks to access shared resources, e.g., if the service API *GetResource(res1)* is invoked by *running* task, and the priority of the task is lower than the ceiling priority of the resource *res1*, the priority of the task will be raised to the ceiling priority of the resource *res1*, and the priority of the task will be reset to the priority before requiring the resource *res1* when *ReleaseResource(res1)* is invoked by the task. Note that, the ceiling priority of a shared resource is lower than the lowest priority of all tasks that do not access the resource, and it is higher than the priorities of all tasks that access the resource.

2.2 OSEK/VDX Application and Its Execution Characteristics

An application developed based on OSEK/VDX OS consists of two files, one is the source file, and the other is the configuration file. The source file which can be developed by C language is used to present the concrete behaviors of the application. The configuration file is used to define tasks, synchronization events and shared resources. In the configuration file, the attribute *AUTOSTART* is used to set the initial

state of tasks. If the attribute *AUTOSTART* of a task is set to be *TRUE*, the task starts from *ready* state in the initial state (it will be inserted into ready queue according to the priority of the task). Otherwise, the task starts from *suspended* state. The attribute *SCHEDULE* is used to indicate the scheduling type. If the attribute *SCHEDULE* of a task is set to be *FULL*, the task can be preempted by higher priority tasks. Otherwise, the task cannot be preempted by higher priority tasks. A simple OSEK/VDX application without synchronization events and shared resources is shown in Fig. 2. As to clearly comprehend the execution characteristics of OSEK/VDX applications, an example is symbolically executed in this part.

Motivating example: In the motivating application shown in Fig. 2, only the attribute *AUTOSTART* of *t1* is set to be *TRUE*. Thus, *t1* will be firstly moved to *running* state by scheduler and then executed in the initial state. As shown in Fig. 3, when the service API *ActivateTask(t2)* is invoked by *t1* (the branch ($b \geq a$) is chosen to symbolically execute), scheduler will be loaded to respond to the service API (task *t2* is moved from *suspended* state to *ready* state). At this moment, the running task *t1* will be preempted by *t2* since the priority of *t2* is higher than *t1* and the attribute *SCHEDULE* of *t1* is set to be *FULL* (the context switch happens after the service API *ActivateTask(t2)*). Currently, task *t2* gets run-unit to run, and goes to *suspended* state when the service API *TerminateTask()*[†] is invoked. When task *t2* terminates itself using service API *TerminateTask()*, scheduler will dispatch task *t1* to *running* state. Then, *t1* continues its executions from the preempted point, and task *t4* is activated by *t1* using service API *ActivateTask(t4)* (the context switch will not

[†]*TerminateTask()* is used to terminate the executions of a task, and the terminated task will be moved from *running* state to *suspended* state by scheduler. If the running task is terminated, scheduler then dispatches the head task in the ready queue to run when ready queue is not empty.

Fig. 4 The framework of the translation approach

As to successfully translate the application which holds loops into sequential program and avoid the same sub-paths to be poured into the sequential program, in our approach we develop a directed graph constructor as a simulator to construct a directed graph instead of the execution tree for representing the sequential program of the OSEK/VDX application.

3.1 The Overview of the Translation Approach

The framework of our approach is shown in Fig. 4. As to automatically translate an OSEK/VDX application into the corresponding sequential program, in the first step we develop a front-end interpreter to interpret the behaviors of tasks within the application into corresponding control flow graphs (CFGs) based on the C intermediate language (CIL) [19]. Currently, the interpreter can accept the main characteristics of C programming language except pointer, struct and function calls. In the second step, the directed graph constructor (DGC) is employed to construct a directed graph for representing the sequential program of OSEK/VDX application. Particularly, an OSEK/VDX OS model is embedded in the DGC (translation algorithm flat) to respond to the invoked service APIs and compute the running task when the directed graph generator meets a service API in the process of constructing directed graph. Moreover, two optimization strategies named cut-task and acc-sch in the scheduling level for cutting the unnecessary states and accelerating the scheduling of the tasks with assertions are integrated in the embedded OS model. In the third step, the constructed directed graph is compiled into the object language according to the syntax of input language of back-end model checker. In the last step, the model

Fig. 5 The task CFGs of the motivating example

checker such as Spin will be employed to verify the obtained model.

3.2 Task CFG

DEFINITION 1 (CFG of task): The CFG of a task is a tuple $\Omega^{tid} = (N^{tid}, n_0^{tid}, n_e^{tid}, \Sigma^{tid}, R^{tid})$. Where, tid is the identifier of tasks. N^{tid} is the set of locations, $n_0^{tid} \in N^{tid}$ is the start location, $n_e^{tid} \in N^{tid}$ is the end location. Σ^{tid} is the set of statements of task tid , the expression of a statement $\alpha \in \Sigma^{tid}$ is as follows:

$$\alpha ::= condition | assignment | goto | assertion | API$$

$R^{tid} \subseteq N^{tid} \times \Sigma^{tid} \times N^{tid}$ is the set of directed edges labelled by task statements, and (n, α, n') represents an element of R^{tid} . E.g., the CFGs for the motivating example are shown in Fig. 5.

3.3 OSEK/VDX OS Model

In OSEK/VDX OS standard, as to let developers easily and confidently implement an OSEK/VDX OS, all of the functions for responding to service APIs are specifically stated. In our paper, based on the stated functions, as shown in Fig. 6, we construct an abstract OS model which is a combination model of scheduler module, synchronization event process module and shared resource process module in DGC to respond to the service APIs invoked from application.

DEFINITION 2 (OS model): The abstract OS model is a tuple $OS = (N, n_0, n_c, F, R, D)$, where N is the set of nodes, n_0 is the start node, n_c is the end node. F is the set of API responding functions. $R \subseteq N \times F \times N$ is the set of transitions. $D = \{runTask, readyQueue, suspendList, waitList, evtBitArray, resAccessList\}$ is the set of data structures. Here, $runTask$ is a variable used to store the tid of running task (tid is the identifier of tasks). In the OSEK/VDX OS several tasks can share a same priority, the $readyQueue$ is composed of queues with different priorities, used to store

Fig. 6 OSEK/VDX OS model

the *tids* of tasks in the *ready* state. The data structures *suspendList* and *waitList* are used to store the *tids* of tasks in the *suspended* state and *waiting* state, respectively. In OSEK/VDX application the extended tasks can hold synchronization events, *evtBitArray* is a matrix used to store the event states of extended tasks (*eid* is the identifier of events). Moreover, tasks within the OSEK/VDX application can access shared resources according to the priority ceiling protocol, *resAccessList* is composed of lists used to indicate the state of resources accessed by tasks (*rid* is the identifier of resources). In addition, in the constructed OS model, the trace ϖ which starts from node n_0 and ends at node n_3 represents the responding behaviors of the OS model for an invoked service API.

In order to call the constructed OS model to respond to the invoked service APIs and compute running task in the process of generating directed graph, we define two interface functions *StartTask()* and *RespondAPI(API)* for the constructed OS model. Where, the interface function *StartTask()* is used to request OS model to dispatch the head task in *readyQueue* to *running* state, the interface function *RespondAPI(API)* is used to call the corresponding trace ϖ to respond to the invoked service APIs and compute the data within *D* of OS model. For instance, if the interface function *RespondAPI(API)* is called to respond to the invoked service API *ActivateTask(t1)*, the function *ActivateTask(t1)* of OS model is firstly called to active task *t1*, and then the function *Preempt()* is called to judge whether the rescheduling point happens (if the rescheduling point happens, the currently running task will be moved to *readyQueue*, and *runTask* will be set to null by the *Preempt()* function). Finally, the function *Start()* is called to dispatch the head task in *readyQueue* to run if *runTask* is null.

3.4 Directed Graph Constructor

In the execution tree method, as to accurately describe the executions of an OSEK/VDX application, we explore all of the execution paths to show the different task execution sequences in different branches. Actually, to the general branches without service APIs, we do not need to explore these branches, instead, we can directly use a directed graph to represent these branches. For example, in the execution tree shown in Fig. 3, we can use the CFG of task *t2* illustrated in Fig. 5 to replace the part from ③ to ④. However, to the branches with service APIs, we cannot directly use a directed graph to replace them, because the different service APIs locating at different branches will lead to different data in the data structures *D* of OS model, and the different data within *D* will result in the different task execution sequences in the different branches. As to reduce the size of the sequential program as much as possible, in our approach, we will use one sub-path to represent the equivalent sub-paths (if two sub-paths represent the same behaviors of tasks and hold the same data within *D* of OS model, these two sub-paths are equivalent). E.g., in the execution tree shown in Fig. 3, we can just use the sub-path from ⑤ to ⑥ in the execution path π_1 to represent the equivalent sub-paths in the execution paths π_2 and π_3 .

Based on this idea, we develop a directed graph constructor (DGC) to generate a directed graph instead of execution tree for representing the sequential program of the OSEK/VDX application. The definition of generated directed graph is shown in DEFINITION 3. The details about how to construct the directed graph are stated in Alg. 1.

DEFINITION 3 (directed graph): The directed graph is a tuple $\mathcal{G}=(V, v_0, v_e, E)$. Where, *V* is the set of nodes, and a node $v \in V$ is a tuple $v=(pcs, osd)$, here $pcs=[n^1, \dots, n^m]$ is an array used to record the current locations of tasks $t_1, \dots, t_m$ (*m* is the number of tasks), *osd* is a set of values used to store the data within *D* of OS model. $v_0 \in V$ is the start node, and $v_e \in V$ is the end node. $E \subseteq V \times \bigcup \Sigma^{tid} \times V$ is the set of directed edges labelled by task statements.

In Alg. 1, Θ which is a set is employed to store the transitions of directed graph, where an element γ of Θ is a tuple $\gamma=(v, \alpha, v')$ (here, $v' \in V$ is the successor node of node $v \in V$, $\alpha \in \bigcup \Sigma^{tid}$ is the task statement). Moreover, for two elements $\gamma_i=(v_i, \alpha_i, v'_i)$ and $\gamma_j=(v_j, \alpha_j, v'_j)$, if and only if $v_i = v_j$ and $\alpha_i = \alpha_j$ and $v'_i = v'_j$, γ_i is equal to γ_j . Note that, the API statement is replaced with *goto* statement in the process of constructing directed graph. The structure of Alg. 1 is as follows,

- the fragment (1-3) is the initial part.
- the fragment (5-10) is used to construct the start transition of directed graph.
- the fragment (13-21) is used to compute the new transitions of directed graph.
- the fragment (29-37) is used to call the interface function *RespondAPI(API)* of OS model to respond to the

Algorithm 1 : Directed Graph Constructor

Input: task CFGs $\Omega^1, \Omega^2, \dots$, and configuration file of application
Output: directed graph

- 1: initialize D of OS model with configuration file
- 2: initialize $pcs := [n_0^1, \dots, n_0^m]$ with task start locations
- 3: $i := 1, j := 0$, where i and j are the index of nodes
- 4:
- 5: call the interface function $StartTask()$ of OS model to compute the running task
- 6: $D \rightarrow osd$, where the operator “ $\rightarrow$ ” represents mapping the data within D into osd
- 7: $v_0 := (pcs, osd)$
- 8: $v_1 := v_0$
- 9: $\gamma := (v_0, start, v_1)$ and set γ as unexplored element
- 10: $\Theta := \Theta \cup \gamma$
- 11:
- 12: **while** $\exists \gamma \in \Theta$ is an unexplored element **do**
- 13: $(v, a, v') := \gamma$, and set γ as *explored* element
- 14: $osd := osd$ within node v'
- 15: $pcs := pcs$ within node v'
- 16: $j :=$ the index of node v'
- 17: $tid := runTask$ within osd
- 18: **if** $tid = null$ **then**
- 19: update the index of v' of γ with e , goto 12
- 20: **end if**
- 21: $\Delta := \{(n, a, n') \in R^{tid} \text{ of } \Omega^{tid} | n = pcs[tid]\}$
- 22:
- 23: **for all** $(n, a, n') \in \Delta$ **do**
- 24: $i++$
- 25: $v_j := v'$
- 26: $\alpha :=$ the task statement a of (n, a, n')
- 27: $pcs[tid] :=$ the target location n' of (n, a, n')
- 28:
- 29: **if** α is an API **then**
- 30: **if** α is $TerminateTask()$ or $ChainTask()$ **then**
- 31: $pcs[tid] :=$ the start location n_0 of Ω^{tid}
- 32: **end if**
- 33: $osd \rightarrow D$, where the operator “ $\rightarrow$ ” represents mapping the data within osd into D of OS model
- 34: **call the interface function ResponseAPI(α) of OS model to respond to the service API α and compute the data within D of OS model**
- 35: $D \rightarrow osd$
- 36: $\alpha := goto$
- 37: **end if**
- 38:
- 39: $v_i := (pcs, osd)$
- 40: $\gamma' := (v_j, \alpha, v_i)$
- 41:
- 42: **if** $\exists \gamma'' \in \Theta = \gamma'$ **then**
- 43: $j :=$ the index of node v within γ''
- 44: update the index of v' of γ with j
- 45: goto 12
- 46: **end if**
- 47:
- 48: set γ' as *unexplored* element, $\Theta := \Theta \cup \gamma'$
- 49: **end for**
- 50: **end while**
- 51: **return**

invoked service API and compute the data within D of OS model.

- the fragment (42-46) is used to avoid redundant transitions to be inserted into the directed graph.

Based on the DGC, we can construct a directed graph to represent the sequential program of an OSEK/VDX application, e.g., the directed graph for the motivating example has been illustrated in Fig. 7. In addition, the relationships between statements of tasks can clearly specified by the constructed directed graph, we thus can easily compile the directed graph into the input language of back-end model checker such as Promela. A similar work for translated finite state machine into systemC has been presented in paper [20].

There are three advantages in our approach, e.g., (i) the constructed directed graph can accurately reflect the execu-

Fig. 7 The directed graph for the motivating example

Fig. 8 The new directed graph for the motivating example with loop

tions of the OSEK/VDX application, since the OSEK/VDX OS model is employed to respond to the invoked service APIs and compute the running task; (ii) the redundant sub-paths caused by branches will not be poured into the sequential program, because we just use one sub-path to represent the same sub-paths. (iii) loops within tasks can be easily translated into the sequential program, because we can use cycles to represent them in the directed graph, e.g., for the motivating example, if we use a loop to replace the branches within task $t2$, as shown in Fig. 8, our approach will construct a new directed graph with a cycle to represent the sequential program of the modified motivating example.

However, there is a limitation in our approach, that is, our approach does not support the multi-activation of tasks, which is caused by the loops with APIs. For instance, a task is currently in *readyQueue* and it is multiply activated by the running task using a loop. According to the OSEK/VDX specification, when the task is terminated, it will be directly move to *readyQueue* until the activation times of the task have been used up. However, the task will be activated one time in our approach. This is because, in our approach we do not judge the execution times of the loop and the loop just be symbolically executed on time, that is, we do not know how many times the task is activated by the loop. As to solve this limitation, before using DGC to generate the directed graph, we can set a bound for the loop with APIs, and unfold the loop according to the set bound. The idea benefits from bounded model checking [15], [16]. For instance, to the example shown in Fig. 9, we can unfold the loop two times when we set loop bound to 2. In particular, in order to judge whether the loop is unfolded enough or not, we can insert an assertion which holds the negative condition of the loop in the end of unfolded loop body.

3.5 Optimization Strategies: Acc-Sch and Cut-Task

According to the execution characteristics of the OSEK/VDX applications, we have found, the invoked service APIs will dynamically change the data within OSEK/VDX OS (such as the data in the ready queue), and the changed data will lead to the different task execution sequences in the future. However, there is a particular service API *TerminateTask()* that will not lead to the different task execution sequences in the future, because the service API *TerminateTask()* is only used to terminate a task. In other words, the states of other tasks will not be changed by the service API *TerminateTask()*. Therefore, if a task just holds *TerminateTask()* API, the task execution sequence indicated by the ready queue will not be changed by this task. In our paper, we name this type of tasks as pure task. Moreover, based on the pure task, two optimization strategies are proposed in our paper. The first strategy named acc-sch is used to accelerate the speed of bug hunting by accelerating the scheduling of the task with assertions, the second strategy named cut-task is used to reduce the size of constructed directed graph by cutting the unnecessary task states.

acc-sch strategy: The key idea of acc-sch strategy is to dispatch the pure task which holds assertions to be firstly executed for speeding up the bug hunting of back-end model checker, which benefits from the partial order reduction (POR) [21]. As shown in Fig. 10, there are two pure tasks *t1* and *t2* in the same queue. We assume that, task *t1* does not hold assertions, task *t2* holds an assertion, and tasks *t1* and *t2* are independent. As to accelerate the speed of bug hunting, we can swap the execution order of these two tasks in order to make task *t2* to be firstly executed. In the process of constructing directed graph, the strategy can be called by OS model to swap the execution order of pure tasks in order to accelerate the scheduling of the tasks with

Fig. 9 Unfolding the loop which holds service API according to the loop bound

Fig. 10 The example for acc-sch strategy

assertions. The swap condition for two tasks is that, if two adjacent tasks locating at the same priority-queue are independent and these two tasks are the pure task, the execution order of these two tasks can be swapped in the scheduling queue. The independent condition of two tasks is shown in DEFINITION 4.

DEFINITION 4 (independence of two tasks): two tasks $t=\{R, W\}$ and $t'=\{R', W'\}$, if and only if the accessed variables satisfy the condition $R \cap W' = \emptyset$ and $R' \cap W = \emptyset$ and $W \cap W' = \emptyset$, tasks t and t' are independent, where R and W are the sets of read-variables and write-variables of task, respectively.

cut-task strategy: In addition to the acc-sch strategy, based on the pure tasks, we also propose an optimization strategy named cut-task to cut the unnecessary task states from constructed directed graph in order to make back-end model checker more scalable. The key idea of the cut-task strategy is that, if a pure task does not hold assertions and it is independent with other tasks, the task can be regarded as an unnecessary task and its behaviors will not be mapped in the directed graph in the sequentialization process. The process for detecting the unnecessary tasks from application is as follows,

- step 1** initialize two sets P and Q , where P is the unnecessary task set.
- step 2** for each task t in application, do: if task t is the pure task and the task does not hold assertions, $P = P \cup \{t\}$; otherwise, $Q = Q \cup \{t\}$.

Fig. 11 The cut directed graph for the motivating example

step 3 if $\exists t \in P$ and $\exists t' \in Q$ are not independent, then $Q = Q \cup \{t\}$, $P = P \setminus \{t\}$. Repeat step 3 until there is no new task that is put into Q .

Based on the cut-task strategy, in the process of constructing directed graph, as to cut the states of an unnecessary task from directed graph, OS model will directly move the task to suspended state when it is the currently running task. E.g., in the motivating example, the pure task t_2 is an unnecessary task since it is independent with other tasks and does not hold assertions. In the sequentialization process, in order to cut the states of t_2 , OS model will directly move task t_2 to suspended state when it is the currently running task. Furthermore, as shown in Fig. 11, for the motivating example, cut-task strategy will cut 4 states from the original directed graph illustrated in Fig. 7.

4. Implementation and Experiments

4.1 Implementation

We have developed a prototype tool named autoC according to the proposed approach. The tool consists of four modules, implemented on the Visual Studio 2010 with 4000 lines of C++ code. The first module is to interpret task behaviors of application into the corresponding CFGs (the interpreter for now can accept the main characteristics of C programming language except pointer, struct and function calls). The second module is to extract the configuration data from configuration file of application. The third module is to implement the functionality of DGC. The last module is to compile the constructed directed graph into the object language according to the syntax of input language of back-end model checker.

4.2 Experiments and Discussion

An OSEK/VDX application usually consists of tasks, APIs, loops, synchronization events and shared resources. In the

experiments, as to comprehensively investigate the effectiveness of our approach on the realistic applications, the applications which hold different task number, API number and loop number are selected as our benchmarks (most of the benchmarks are taken from the papers of synM checking method and EPG technique). Moreover, as to really represent the execution behaviours of an OSEK/VDX application, the non-preemptive scheduling behaviour (e.g., msgp1-msgp4 shown in Table 1), full-preemptive scheduling behaviour (e.g., token1-token4), mix-preemptive scheduling behaviour (e.g., rw_var1-rw_var4), synchronous behaviour (e.g., sync1-sync4), and accessing shared resource behaviour (e.g., acc_res1-acc_res4) are also taken into account in the selected benchmarks. Moreover, in the selected benchmarks, assertion as a checking property is verified in the experiments for showing the performance of our approach.

In addition, in order to impartially demonstrate the practicality of our approach, the well-known Spin model checker is selected as our back-end model checker. Moreover, the synM checking method and EPG technique are considered as our comparison objects. All of the experiments are conducted on the Intel Core(TM)i7-3770 CPU with 32G RAM, and we set the time limit and memory limit to 600 seconds and 1GB, respectively. The benchmarks used in the experiments and the prototype tools corresponding to the synM checking method and EPG technique are available at <http://www.jaist.ac.jp/~s1220209/autoC.htm>. Note that, in order to make all checking methods used in the experiments check the same state space, in the autoC+Spin method the max depth is set to “20,000,000”. In the EPG technique, the max depth is set to “20,000,000”, and the loop bound is set to 10. In the synM checking method, the “C compiler” of Spin is set to “-DVECTORSZ=16384 -DBITSTATE”, and the max depth is set to “20,000,000”.

The experiment results have been listed in Table 1. In the result table, # t is the number of tasks, # l is the number of loops, #API is the number of times of invoked APIs, # s is the number of explored states. “MB” is the memory consumption measured in Mbyte, “time” is the time consumption measured in second. “M.O.” and “T.O.” stand for memory out and time out. “optimization⁺” and “optimization⁻” represents that autoC disables and enables the optimization strategies acc-sch and cut-task in the sequentialization process, respectively.

Based on the shown results, we can find, (i) in all of the conducted experiments, autoC+Spin method will check lesser states than synM checking method and EPG technique, especially when the cut-task and acc-sch strategies are effective to cut unnecessary task states and accelerate the scheduling of the tasks with assertions (e.g., lines 4, 8, 12, 15 and 20); (ii) in contrast with synM checking methods and EPG technique, autoC+Spin method can successfully verify the given benchmarks which hold a lot of loops, tasks and APIs with shorter time and smaller memory. Moreover, the experiment results indicate that our approach can be considered as a practical method to check the realistic

Table 1 Comparison: synM, EPG and autoC+Spin

benchmark	size			original version						autoC		Spin					
				synM			EPG					optimization [†]			optimization [†]		
	#t	#l	#API	#s	MB	time	#s	MB	time	MB	time	#s	MB	time	#s	MB	time
1 msgp1.safe	6	0	11	4652	17.7	0.21	89	2.17	0.29	3.19	0.18	36	0.02	0	36	0.02	0
2 msgp2.bug	8	0	14	25253	81.1	0.71	55	2.18	0.35	3.64	0.26	38	0.02	0.01	38	0.02	0.01
3 msgp3.safe	10	0	17	103835	303	2.67	157	2.19	0.49	4.11	0.30	60	0.04	0.01	60	0.04	0.01
4 msgp4.bug	18	0	35	-	-	T.O.	145	2.22	0.57	5.72	0.60	99	0.09	0.02	56	0.04	0.01
5 token1.safe	4	4	61	20501	2.45	0.10	2470	2.26	43.4	2.96	0.14	1008	0.50	0.11	1008	0.50	0.11
6 token2.bug	6	6	101	34371	126	1.12	2283	2.23	139	3.34	0.19	1624	0.93	0.16	1624	0.93	0.16
7 token3.safe	9	9	161	46990	138	1.26	6417	2.41	192	4.05	0.29	2607	1.79	0.27	2607	1.79	0.27
8 token4.bug	13	13	241	-	M.O.	-	-	-	T.O.	5.14	0.45	3904	2.78	0.39	2584	1.71	0.25
9 rw.var1.safe	5	5	101	13541	54.7	0.44	3259	2.30	78.4	3.21	0.18	1308	0.75	0.13	1308	0.75	0.13
10 rw.var2.safe	9	9	161	-	M.O.	-	6459	2.41	450	4.22	0.30	2588	1.71	0.26	2588	1.71	0.26
11 rw.var3.safe	13	13	241	-	M.O.	-	-	-	T.O.	5.23	0.45	3868	2.56	0.38	3868	2.56	0.38
12 rw.var4.safe	13	12	13	-	M.O.	-	940	2.34	13.4	4.78	0.41	390	0.24	0.04	198	0.11	0.02
13 sync1.safe	5	4	14	2443	11.6	0.15	336	2.22	4.02	3.16	0.18	134	0.11	0.02	134	0.11	0.02
14 sync2.safe	8	7	23	24159	86.8	0.78	585	2.27	7.14	4.01	0.26	230	0.14	0.02	230	0.14	0.02
15 sync3.bug	11	10	32	210841	612	5.37	458	2.31	10.7	4.53	0.40	322	0.02	0.03	226	0.02	0.02
16 sync4.safe	12	11	42	382329	998	9.62	917	2.33	11.9	5.11	0.45	358	0.19	0.03	358	0.20	0.03
17 acc.res1.safe	2	3	4	13907	56.5	0.46	3491	2.32	88.8	3.41	0.20	1308	0.75	0.13	1308	0.75	0.13
18 acc.res2.safe	9	9	320	-	M.O.	-	6779	2.41	509	4.63	0.34	2588	1.71	0.26	2588	1.71	0.26
19 acc.res3.safe	13	13	480	-	M.O.	-	-	-	T.O.	5.84	0.61	3868	2.56	0.38	3868	2.56	0.38
20 acc.res4.bug	12	12	25	-	M.O.	-	552	2.13	13.9	4.98	0.44	389	0.24	0.03	197	0.11	0.02

OSEK/VDX applications, since the back-end model checker Spin just spends a few cost in the verification.

Comparison to synM: According to the conducted experiments, we can find that the synM checking method is failed to verify the benchmarks which contain a lot of tasks and service APIs (e.g., lines 4, 12 and 20). This is because, in the synM checking method the checking model is a combination of OSEK/VDX OS model and application model. Therefore, the synM checking method will not only check the behaviors of tasks but also verify the OS model in the verification, which will limit the capability of synM checking method. Compared with synM checking method, the autoC+Spin method can successfully verify these benchmarks with lower costs, this is because that the back-end model checker Spin just verify the task behaviors in the verification. Moreover, in the synM checking method, all of the states corresponding to both tasks and OS model will be stored in the memory in the checking process. In contrast with synM checking method, in the autoC+Spin checking method, only task states are checked and stored in the memory in the verification. These efforts make autoC+Spin method more efficient than synM checking method on handling the same benchmarks. Based on the comparison results, it can be seen that, our autoC can significantly increase the capability of Spin model checker in checking the OSEK/VDX applications which contain a lot of tasks and service APIs.

Comparison to EPG: According to the conducted experiments, we can find, although the EPG technique can handle the complex benchmarks which hold a lot of tasks and APIs, it will take longer to verify the given benchmarks which contain a lot of loops, and even runs out of time (e.g., lines 8, 11 and 19). This is because, in the EPG technique, as to accurately describe the different task execution sequences in different branches, the transition system of application is constructed based on the execution paths. Therefore, when the benchmark holds a lot of loops and branches, EPG technique will check a large number of execution paths and a large number of the same sub-paths will be repeatedly verified in the verification, which will slow down the performance of EPG technique. In contrast with EPG technique, in the autoC+Spin method, the same sub-paths will not be

repeatedly verified by Spin, since the same sub-paths have been cut by autoC in the sequentialization process. Based on the comparison results, it can be seen that, our autoC can significantly improve the effectiveness of Spin model checker in checking the OSEK/VDX applications which hold a lot of loops and branches.

5. Related Work

With the development of OSEK/VDX OS standard, OSEK/VDX has been widely applied in the development of vehicle-mounted OS. Currently, how to ensure the reliability of the developed OSEK/VDX OS and its applications is becoming a challenge for developers with the continuously increasing complexity in the development process. To the scope of checking developed OSEK/VDX OS, there are some invaluable methods, e.g., Chen and Aoki have proposed a method [22] to generate the highly reliable test-cases for checking whether the developed OS conforms to the OSEK/VDX OS standard based on the Spin model checker. As to support an environment of OSEK/VDX OS for the model checking, an UML-based method for producing Promela scripts of OSEK/VDX OS is proposed in paper [23]. In addition, for the Trampoline [24] which is an open source RTOS developed based on the OSEK/VDX OS standard, Choi presented a method [25] to convert the Trampoline kernel into formal models, and in the method an incremental verification approach is proposed to carry out the verification. Furthermore, a CSP-based approach for checking the code-level OSEK/VDX OS is addressed in the paper [26]. All of these related works are different from our work, because our approach focuses on the developed OSEK/VDX applications.

To the developed OSEK/VDX applications, the paper [27] has proposed a method to check the timing property based on the UPPAAL. In addition, in order to use Spin model checker to check the safety property of developed OSEK/VDX applications, in our previous work [10] a synchronization model (synM) is used to simulate the executions of the OSEK/VDX application. As mentioned in the experiments, although the method can precisely verify the OSEK/VDX applications using Spin, the reported results show that this method cannot handle the complex programs,

because too many details about OS model are poured into the checking model. As to avoid the OS model to be checked in the verification, in the paper [13] we present a new approach named EPG to check OSEK/VDX applications based on the SMT-based bounded model checking technique. Our approach in this paper is similar to EPG technique in embedding OS model in algorithm flat, but our target is to translate the OSEK/VDX application into sequential program, which is different from EPG technique.

In the field of translating multi-threaded software into sequential program, there have been several works, e.g., in paper [7] Cimatti et al. propose an approach to translate systemC program into the sequential program. In addition, the paper [28] also shows an approach to translate the multi-threaded software that conforms to the POSIX standard [29] into a non-deterministic sequential program. However, these existing methods focus on the non-deterministic scheduler, they cannot be used to implement the sequentialization process of OSEK/VDX applications since the deterministic scheduler is adopted by OSEK/VDX OS to dispatch tasks. The technique shown in our paper is different from the existing works.

6. Conclusion and Future Work

In this paper, we presented an approach that can automatically translate OSEK/VDX applications into sequential programs for enabling existing model checkers to verify OSEK/VDX applications. Moreover, two optimization strategies in the scheduling level for cutting the unnecessary states and accelerating the speed of bug hunting were also proposed in this paper. We have evaluated our approach based on a series of experiments. The experiment results indicate, (i) our approach can accurately translate the OSEK/VDX applications into sequential programs; (ii) based on the sequentialization process of our approach, the back-end model checker Spin can efficiently verify the given applications with less cost in terms of states, time and memory compared with the related checking methods synM and EPG technique; (iii) the state reduction and acceleration scheduling strategies are useful to cut the unnecessary task states from the sequential program and accelerate the speed of bug hunting in the verification.

In the future, several works will be carried out based on the shortcomings of the current work. First, we will improve our approach to accept more complex C programming language such as `pointer`, `struct` and function calls. Second, we will extend our approach to process the applications with interruptions. Finally, we will apply our approach to handle the realistic OSEK/VDX applications.

References

- [1] H. Zhang, T. Aoki, and Y. Chiba, "Yes! You Can Use Your Model Checker to Verify OSEK/VDX Applications," accepted by IEEE International Conference on Software Testing, Verification and Validation (ICST), Graz, April 2015.
- [2] J. Lemieux, Programming in the OSEK/VDX Environment, CMP, Suite 200 Lawrence, KS 66046, USA, 2001.
- [3] OSEK/VDX Group, "OSEK/VDX operating system specification 2.2.3," <http://portal.osek-vdx.org/>, accessed May. 10. 2015.
- [4] E.M. Clarke, O. Grumberg, and D.E. Long, "Model Checking and Abstraction," *ACM Trans. Programming Languages and Systems (TOPLAS)*, vol.16, no.5, pp.1512–1542, Sept. 1994.
- [5] E.M. Clarke, E.A. Emerson, and J. Sifakis, "Model Checking: Algorithmic Verification and Debugging," *Commun. ACM*, vol.52, no.11, pp.74–84, Nov. 2009.
- [6] Z. Yang, C. Wang, A. Gupta, and I. Franjo, "Model Checking Sequential Software Programs Via Mixed Symbolic Analysis," *ACM Trans. Design Automation of Electronic Systems*, vol.14, no.1, pp.1–26, Jan. 2009.
- [7] A. Cimatti, A. Micheli, I. Narasamya, and M. Roveri, "Verifying SystemC: A software model checking approach," *FMCAD*, pp.51–59, Oct. 2010.
- [8] L. Cordeiro and B. Fischer, "Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking," *ICSE*, vol.3, no.9, pp.331–340, May 2011.
- [9] SystemC standard, "IEEE 1666: SystemC language Reference Manual," 2005.
- [10] H. Zhang, T. Aoki, and Y. Chiba, "A Spin-based Approach for Checking OSEK/VDX Applications," 3rd International Workshop FTSCS in ICFEM, pp.187–202, Nov. 2014.
- [11] G.J. Holzmann, The Spin Model Checker: Primer and Reference Manual, Lucent Technologies Inc., Bell Laboratories, Boston, USA, Sept. 2003.
- [12] E.M. Clarke, W. Klieber, et al., "Model Checking and the State Explosion Problem," *Tools for Practical Software Verification*, vol.7682, pp.1–30, 2012.
- [13] H. Zhang, T. Aoki, H.-H. Lin, M. Zhang, Y. Chiba, and K. Yatake, "SMT-based Bounded Model Checking for OSEK/VDX Applications," 20th APSEC, pp.307–314, Dec. 2013.
- [14] H. Zhang, T. Aoki, K. Yatake, M. Zhang, and H.-H. Lin, "An Approach for Checking OSEK/VDX Applications," 13th QSI, pp.113–116, July 2013.
- [15] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *STTT*, vol.11, no.1, pp.69–83, 2009.
- [16] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in Computers*, vol.58, no.11, pp.117–148, 2003.
- [17] Uppsala University, Sweden and Aalborg University, "UPPAAL," <http://www.uppaal.org/>
- [18] A. Burns and A. Wellings, Real-Time Systems and Programming Languages (4th Edition), Addison Wesley Longman, New York, NY, USA, Aug. 2009.
- [19] G. Necula, "CIL," <http://kerneis.github.io/cil/>, accessed May. 10. 2015.
- [20] T.H. Khan, A. Habibi, et al., "A Tool for Converting Finite State Machines to SystemC," Technical Report, Concordia University, Canada, 2007.
- [21] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems, PhD thesis, University of Liege, Computer Science Department, Nov. 1994.
- [22] J. Chen and T. Aoki, "Conformance Testing for OSEK/VDX Operating System Using Model Checking," 18th APSEC, pp.274–281, 2011.
- [23] K. Yatake and T. Aoki, "Automatic Generation of Model Checking Scripts Based on Environment Modeling," *SPIN*, pp.58–75, 2010.
- [24] IRCCyN group, "Trampoline," <http://trampoline.rts-software.org/>, accessed May. 10. 2015.
- [25] Y. Choi, "Safety Analysis of Trampoline OS Using Model Checking: An Experience Report," *Software Reliability Engineering (ISSRE)*, pp.200–209, Nov. 2011.
- [26] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi, "Modeling and Verifying the Code-Level OSEK/VDX Operating System with

- CSP,” 5th IEEE Trans. Autom. Sci. Eng., pp.142–149, 2011.
- [27] L. Waszniowski and Z. Hanzálek, “Formal verification of multitasking applications based on timed automata model,” *Real-Time Systems*, pp.39–65, 2008.
- [28] O. Inverso, E. Tomasco, B. Fischer, S.L. Torre, and G. Parlato, “Lazy-CSeq: A Lazy Sequentialization Tool for C,” 20th TACAS, pp.398–401, 2014.
- [29] ISO/IEC, “Information technology-Portable Operating System Interface (POSIX) Base Specifications,” Issue 7, ISO/IEC/IEEE 9945:2009, ISO, 2009.

Haitao Zhang is presently a Ph.D. student in School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his M.S. degree from JAIST & Tianjin University (China) in 2012. His current research interests include model checking, software testing, and cyber-physical system (CPS).

Toshiaki Aoki is an associate professor in School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his B.S. degree from Science University of Tokyo in 1994. He received his M.S. and Ph.D. degrees from JAIST in 1996 and 1999 respectively. His current research interests include software engineering, software science, and formal method.

Yuki Chiba is an assistant professor in School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his B.S., M.S. and Ph.D. degrees from Tohoku University in 2003, 2005 and 2008 respectively. His current research interests include program transformation, term rewriting system, and automated theorem proving.