A Formal Semantics of the OSEK/VDX Standard in 𝕂 Framework and Its Applications

Min Zhang[1](✉), Yunja Choi[2], and Kazuhiro Ogata[1]

[1] Research Center for Software Verification, Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan
{zhangmin,ogata}@jaist.ac.jp

[2] School of Computer Science and Engineering, Kyungpook National University, Daegu, South Korea
yuchoi76@knu.ac.kr

Abstract. The OSEK/VDX is an international standard of automobile operating systems. Such systems are safety-critical and require extensive safety analysis and verification. Formal methods have been shown useful and effective to verify the safety of both the OSEK/VDX-based operating systems and applications. Using formal methods requires formal semantics of the OSEK/VDX standard. In this paper, we present a formal semantics of the standard using 𝕂, a rewrite-based formal semantics framework. With the formal semantics, we can (1) verify user-defined applications by model checking, and (2) automatically generate test cases for testing of the OSEK/VDX-based operating systems. Features of the formal semantics are its executability and flexibility. Compared with existing formal semantics of the standard, the formal semantics defined in 𝕂 is more flexible and generic. This work also shows that 𝕂 is not only used for formalizing the semantics of programming languages, but also for automobile operating systems.

1 Introduction

The OSEK/VDX is an international standard of developing automobile operating systems [1]. An automobile operating system is a piece of safety-critical software to manage resources and applications which run on the system to control electrical devices in automobiles. Its safety should be extensively analyzed and verified. To implement an OSEK/VDX-based operating system, the traditional approach is to develop both the kernel and applications following the standard, and compile them together to generate an executable system. The system must be tested extensively for safety [2]. This approach is effort-consuming and prone to errors in that modification to source code usually requires recompilation and testing requires complete suite of test cases, which usually are difficult to build.

This research was supported by Kakenhi 23220002, Japan, and by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2013-H0301-13-5004) supervised by the NIPA(National IT Industry Promotion Agency).

© Springer International Publishing Switzerland 2014
S. Escobar (Ed.): WRLA 2014, LNCS 8663, pp. 280–296, 2014.
DOI: 10.1007/978-3-319-12904-4_16

Our previous work [3] and the work [4] have shown that using formal methods is an effective approach to both safety verification of OSEK/VDX systems and applications, which is complementary to the traditional testing-based approach. Using formal methods requires formal semantics of the OSEK/VDX standard. In this paper, we present an executable formal semantics of the standard, which is defined in $\mathbb{K}$, a rewrite-based formal framework [5]. We choose $\mathbb{K}$ for its executability, flexibility, simplicity and tool-support. $\mathbb{K}$ allows user-defined data types and supports formalization of infinite-state systems. Especially, $\mathbb{K}$ provides tool-support to automatically generate interpreter and state-space explorer based on the defined semantics. Another advantage of using $\mathbb{K}$ is that it does not require extra effort to transform user-defined applications into corresponding formal definition in $\mathbb{K}$ in order to use the formal semantics. In this sense, the formal semantics of the standard in $\mathbb{K}$ is more flexible and generic than those formalized in Promela [6] and NuSMV [4], which have restriction on the number of tasks, resources, events in OSEK/VDX-based operating systems, and also need extra effort to instantiate the semantics with user-defined applications, though the number of tasks, resources and events must be fixed when the formal semantics is used for model checking.

The benefit from this formal semantics is multifold. Firstly, it can be used to model check user-defined OSEK/VDX applications with a fixed number of tasks, resources and events by integrating the formal semantics with the semantics of the language in which the applications are implemented. Secondly, it can be used to generate test cases for the testing of OSEK/VDX-based operating systems. This work shows that $\mathbb{K}$ is also well suited to the formalization of automobile operating systems besides the formalization of semantics of programming languages [7–9].

Organization of the paper: Sect. 2 introduces the background and our overall approach. Sections 3 and 4 describe the OSEK/VDX standard and $\mathbb{K}$. Section 5 shows the formalization of OSEK/VDX in $\mathbb{K}$. Section 6 demonstrates two applications of the formal semantics, i.e., model checking and test case generation. Sections 7 and 8 mention some related work and conclude the paper.

2 Background and Overall Approach

The OSEK/VDX standard is a generic description which is mandatory for any implementation of an OSEK/VDX operating system. It concerns the general description of the strategy and functionality, standardized application programming interface (API), resource management, event mechanism, etc. Figure 1 depicts the traditional process of implementing OSEK/VDX-based operating systems [10]. An OSEK/VDX-based operating system is built out of a kernel which includes basic functionality described in the standard such as scheduler, APIs, etc., and a group of applications which interact with the kernel through APIs. An application includes a configuration of resources, tasks, and events that are defined in OIL (OSEK Implementation Language) and source code for each task of the application. They are compiled together and an executable operating

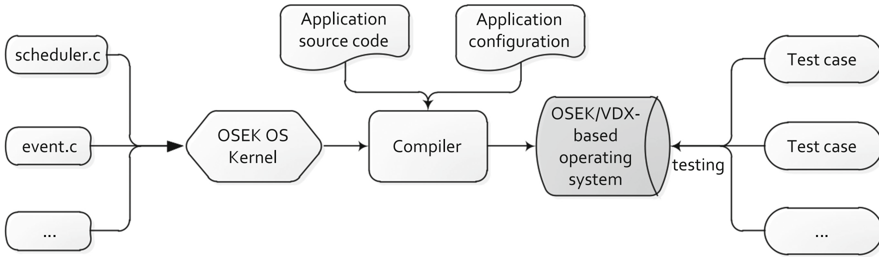

Fig. 1. Traditional approach to the implementation of OSEK/VDX operating systems

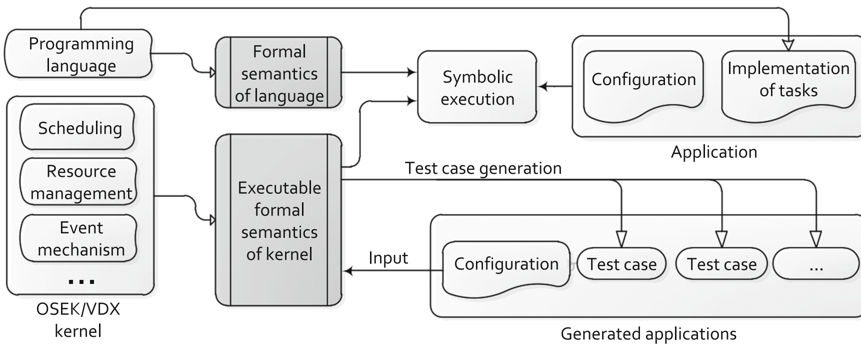

Fig. 2. The framework of our formal approach

system is developed. The system is then extensively tested for safety. There are two major problems with the traditional approach. One is that the whole system must be re-compiled and re-tested due to every change to either the kernel or applications. It is costly in terms of both effort and time. Another problem is that it is prone to errors due to the ambiguity of the standard which in written in natural language and the lack of test cases. A suite of comprehensive test cases are necessary to detect potential errors in a system, but it is not an easy task to build such a suite of test cases.

Using formal methods is an effective means of developing reliable OSEK/VDX-based operating systems, complementary to the traditional one. Figure 2 shows the overview of our formal approach to the verification of OSEK/VDX-based applications and automatic generation of test cases for the testing of OSEK/VDX-based operating systems. The OSEK/VDX standard, including features such as task scheduling, resource management, or event mechanism is formalized. To verify user-defined applications, the semantics of the programming language in which tasks are implemented should also be formalized. User-defined applications can be model checked with the integration of the two formal semantics. By model checking, we can detect potential errors such as deadlock in applications. For test case generation, users only need to provide a configuration of tasks, resources and events, and the constraints that generated test cases should

satisfy, such as the number of APIs for each task. Test cases are automatically generated based on the formal semantics of the kernel. Generated test cases can be used for conformance checking of OSEK/VDX-based operating systems by checking whether the result obtained by running each test case on practical system is the same as the expected one.

3 The OSEK/VDX Standard and OIL

As mentioned earlier, the OSEK/VDX standard generically describes all mandatory requirements of an OSEK/VDX-based operating system. In our current formalization we only consider some fundamental parts in the standard such as task scheduling, resource management, event mechanism, error handling, and leave others such as interruption and real-time feature as future work.

Task and Task Scheduling. Task is the basic building block of an OSEK/VDX application. Multitask is one of the basic requirements of OSEK/VDX-based operating systems. The OSEK/VDX standard specifies two kinds of tasks, i.e., *basic task* and *extended task*. Figure 3 shows the state transitions of basic tasks and extended tasks. A basic task has three states, i.e., *ready*, *running* and *suspended*, while an extended task has a *waiting* state besides the three. The difference between them is that the extended tasks can wait for events during execution by using the system call *WaitEvent*, while the basic tasks cannot. Calling *WaitEvent* may result in a *waiting* state, and the release of the processor. The processor can be reassigned to a lower-priority task without the need to terminate the running extended task.

Tasks are controlled by the scheduler. The scheduler decides on the basis of the task priority which is the next of the *ready* tasks to be transferred into the *running* state. The OSEK/VDX standard provides two scheduling policies, i.e., *full preemptive* and *non-preemptive* scheduling. By full preemptive scheduling, a running task may be rescheduled at any instruction by the occurrence of trigger conditions pre-set by the operating system, such as successful termination of a task, and activating a task. The running task is put into the ready state, as soon as a higher priority task gets ready.

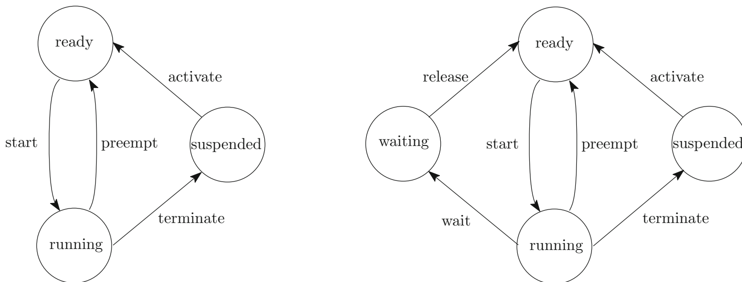

Fig. 3. The state model of basic task (left) and extended task (right)

Resource Management and Priority Ceiling Protocol. Resource management is used to co-ordinate concurrent accesses of several tasks with different priorities to shared resources. It ensures that two tasks can never occupy the same resource at the same time, deadlocks will never occur by use of these resources, and access to resources never results in a *waiting* state.

There are some restrictions when using resources. When occupying a resource, the task should not call some APIs such as *TerminateTask*, which may cause rescheduling after they are called. If a task is occupying multiple resources, these resources must be released in LIFO order.

However, under these restrictions it is possible that a lower-priority task may delay the execution of higher-priority task, which is called *priority inversion*. An example about it can be found in [1]. To avoid priority inversion, OSEK prescribes the OSEK Priority Ceiling Protocol (PCP). The protocol requires that each resource has a *ceiling priority* which is statically assigned at the system generation. Basically, the priority shall be set at least to the highest priority of all tasks that can access that resource. If a task requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task is dynamically raised to the ceiling priority of the resource. If the task releases a resource, the priority of the task is reset to the one before it requires that resource.

Event Mechanism. Tasks in OSEK/VDX-based operating systems are synchronized by events. Events are the criteria for the transition of extended tasks from the *waiting* state to the *ready* state (see Fig. 3). Events are not independent objects, but assigned to extended tasks. An event can be assigned to multiple extended tasks, and each extended task has a definite number of events. It should be statically declared that which events can be assigned to an extended task in OSEK/VDX applications.

When activating an extended task from the *suspended* state, its events are cleared by the system. An extended task goes into *waiting* state when it is running and the events that it is waiting for are not set. The task keeps in the *waiting* state and goes into *ready* state until any event is set by another task. All tasks can set any event of any non-suspended extended task, but only the owner can clear its events. Details about event mechanism can be referred to [1].

OSEK Implementation Language (OIL). OIL is used to configure tasks, resources, events and their relations in an OSEK/VDX-based operating system [10]. Figure 4 shows an example of how to declare resources, events and tasks in OIL. It says that in the corresponding application there is a resource named `r1`, an event named `e1`, and a task named `t1`. Resource `r1` is declared as a standard resource. Event `e1` is declared with a mask as `AUTO`. Event mask is an integer number. If a mask is set `AUTO`, one bit is assigned to it. The statements in the configuration of task `t1` say that the task should be automatically started (put into *ready* state) after the initialization of operation system. The priority of the task is 3, and it is preemptable (indicated by `FULL`). The last two statements in `t1` mean that task `t1` can access resource `r1`, and it has the event `e1`.

```
RESOURCE r1 {
  RESOURCEPROPERTY = STANDARD;
};

EVENT e1 {
  MASK = AUTO;
};
```

```
TASK t1{
  AUTOSTART = true;
  PRIORITY = 3;
  SCHEDULE = FULL;
  RESOURCE = r1;
  EVENT = e1;
};
```

Fig. 4. An example of OSEK/VDX application configuration in OIL

4 The $\mathbb{K}$ Framework

$\mathbb{K}$ is a rewrite-based semantics definitional framework, in which programming languages, calculi, as well as type systems or formal analysis tools can be defined or formalized [5]. A $\mathbb{K}$ definition of a semantics is automatically translated into Maude [11] rewrite theories, which are efficiently executable and can be used for state-exploration by exhaustive behavior analysis such as model checking [11]. The $\mathbb{K}$ framework has been used to formalize some practical programming languages such as C [7], Scheme [9], Python [8]. Some analysis tools have also been defined in $\mathbb{K}$ for type checking and type inference [5].

Semantics is defined in $\mathbb{K}$ by using labeled and potentially nested cell structures and $\mathbb{K}$ (rewrite) rules. The cell structure is called a *configuration*, which is used to represent system or program state. In this paper, we call it $\mathbb{K}$ configuration to differ from the configuration of OSEK/VDX-based applications. There are two types of $\mathbb{K}$ rules: *computational rules*, which count as computational steps, and *structural rules*, which do not count as computational steps. The role of structural rules is to rearrange the configuration so that computational rules can match and apply. They correspond to the equations and rewrite rules respectively in rewriting logic [12].

The formal definition of a programming language in $\mathbb{K}$ can automatically yield an interpreter for the language, and program analysis tools such as a state-space explorer by model checking, with which we can verify programs in that language by exhaustively explore all possible results under the condition that the state space is finite and reasonably small.

5 Formalizing the OSEK/VDX Standard in $\mathbb{K}$

In this section, we explain our approach to formalizing the OSEK/VDX standard in the $\mathbb{K}$ framework[1].

5.1 $\mathbb{K}$ Configuration of the OSEK/VDX

The $\mathbb{K}$ configuration of a running OSEK/VDX-based operating system consists of over 40 nested cells. Figure 5 shows part of them. Each cell has a label. The

[1] Some details are omitted due to the limitation of space. The complete formalization, $\mathbb{K}$ source code and the examples mentioned in Sect. 6 are available at the webpage http://www.jaist.ac.jp/~zhangmin/osek-formal.html.

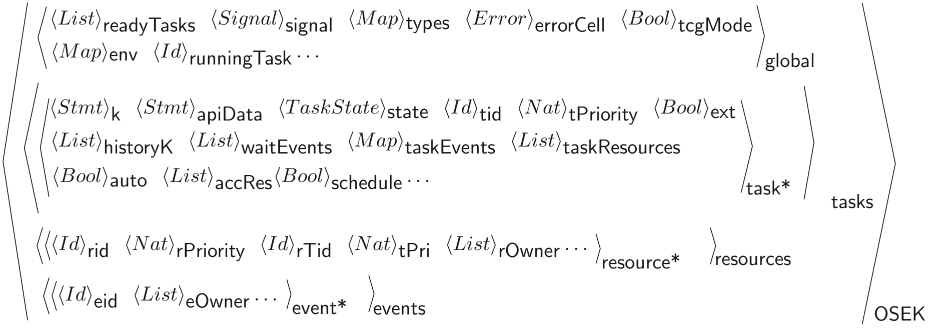

Fig. 5. $\mathbb{K}$ configuration of the OSEK/VDX standard

label ended with $*$ indicates that there can be multiple such cells. A cell that does not have nested cells is a unit cell, storing a term which represents a piece of information of a state. In the brackets is the type of the terms in Fig. 5.

We take the task cell for example. There is both static and dynamic information of a task represented by the nested cells in task. Static information is that configured by users such as task ID in cell tid, priority in cell tPriority, its source code in cell apiData, whether the task is an extended one in cell ext, etc. Dynamic information is that which changes during the execution of operating system such as the next statement to be executed in cell k, the list of events which the task is waiting for in cell waitEvents, the events owned by the task and their status (*set* or *clear*) in cell taskEvents, etc. We do not explain all the cells due to the space limitation. Some will be explained later when needed.

5.2 Formalization of the Scheduler

In our formalization, we only consider *full preemptive* scheduling. As mentioned earlier, the occurrence of trigger conditions such as termination of a task will cause operating system to reschedule tasks. We define a type `Signal` and a constant `schedule` of it. We use a cell with label signal to store the occurrence of such trigger conditions. When there is a signal `schedule` in the signal cell, it indicates that some trigger condition has just occurred. Operating system must first handle it before executing any task. We define a set of $\mathbb{K}$ rules to specify the scheduler. The main one is as follows:

```
rule <signal> schedule => .</signal> <runningTask> I' => I </runningTask>
     <readyTasks> (< I,N >, L) => add2Head(I',N',L) </readyTasks>
     <task> <schedule> FULL </schedule> <state> running => ready </state>
            <tid> I' </tid> <tPriority> N' </tPriority> ... </task>
     <task> <tid> I </tid> <state> ready => running </state> ... </task>
when N >Int N' [transition]
```

The rule specifies how a $\mathbb{K}$ configuration changes before and after scheduling. Before scheduling, there is a signal `schedule` in cell signal. In cell readyTasks there is the list of ready tasks in a descend order by their priority. If there are

two or more ready tasks with the same priority, they are ordered by the time when they get ready. However, the task which is preempted from running is considered as the oldest one among the ready tasks of the same priority [1]. In the cell readyTasks, $<$ I, N $>$, L represents that task I is the oldest one with the highest priority N among the ready tasks. The cells nested in the first task represent that task I' is the present running task with priority N' and it is preemptable, indicated by the value FULL in the cell schedule. If task I has a higher priority than task I', i.e., N >Int N', I' is preempted, and I becomes running. After being preempted, I' is changed into ready state. It is added to the head of the sub-list of the ready tasks which have the same priority as I in the list of ready tasks by function add2Head.

5.3 Formalization of Resource Management

As depicted in Fig. 5, each resource is represented as a cell with label resource, which consists of unit cells for the resource identifier, ceiling priority, and a list of tasks that can access the resource. It also contains two unit cells which are dynamically created when the resource is allocated to a task. The two unit cells are used to store the identifier of the task to which the resource is allocated, and the priority of the task before it gets the resource.

Tasks access resources by two APIs, i.e., *GetResource* and *ReleaseResource*. As mentioned earlier, there are restrictions when accessing resources. Such restrictions together with the Priority Ceiling Protocol should be reflected by the formal semantics of the two APIs. For instance, the main rule defined for *GetResource* is as follows:

```
rule <signal> schedule => .</signal> <runningTask> I' => I </runningTask>
     <readyTasks> (< I,N >, L) => add2Head(I',N',L) </readyTasks>
     <task> <schedule> FULL </schedule> <state> running => ready </state>
            <tid> I' </tid> <tPriority> N' </tPriority> ... </task>
     <task> <tid> I </tid> <state> ready => running </state> ... </task>
when N >Int N' [transition]
```

In the cell k, there is a list of APIs to be executed by task I. The API to be executed next in the list is GetResource(R), where R is a resource ID. The cell tPriority stores the present priority of task I, and the cell accRes stores the list L of resources which task I is accessing. Resource R has a ceiling priority N2. L' in the cell rOwner represents the list of tasks that can access resource R. The condition (following the keyword when) is true when task I can access but is not accessing resource R. When the condition is true, R is allocated to I. According to the Priority Ceiling Protocol, if the present priority of task I is lower than the resource's ceiling priority, it is raised to the ceiling priority, as defined in the cell tPriority. R is added to the head of the list of resources being accessed by I. In the cell resource for R, two cells rTid and tPri are created, storing the task's ID and present priority, i.e., I and N', respectively. The present priority should be stored because when task I releases the resource by *ReleaseResource*, its priority should be reset to the one before it gets the resource. The semantics of *ReleaseResource* can be defined likewise. We omit the details of it in the paper.

5.4 Formalization of Event Mechanism

Each event is represented by an event cell as shown in Fig. 5. An event cell only consists of two sub-cells, storing the static information of the event, i.e., the event name in cell labeled by eid, and a list of owners (tasks) to which the event can be accessed in cell labeled by eOwner. Because event is not an independent object, but is assigned to extended tasks. We declare a new cell with label taskEvents for each extended task, storing the status of events that are assigned to the task. The status of an event is either *set* or *clear*, indicating that the event is set or cleared respectively. We declare a type `EventStatus` and two constants `SET` and `CLEAR` of it to represent the two corresponding status.

There are four APIs associated to events, i.e., *GetEvent*, *SetEvent*, *WaitEvent* and *ClearEvent*, with which tasks can get, set, wait and clear specific events. We take *SetEvent* for example to show how to define its semantics in $\mathbb{K}$. *SetEvent* takes a task ID and an event name. The state of the task specified in the API is transferred to *ready* state, if the event specified in the API is one of the events which the task was waiting for. This can be formalized by the following $\mathbb{K}$ rule:

```
rule <task> <state> running </state> <k> SetEvent(I,E); => . ...</k>
  ...</task> <task> <tid> I </tid> <state> waiting => ready </state>
  <waitEvents> L => . </waitEvents> <tPriority> N </tPriority>
  <taskEvents>... (E |-> (CLEAR => SET)) ...</taskEvents> ...</task>
  <readyTasks> L' => add2Tail(I,N,L') </readyTasks>
  <signal> . => schedule </signal>  when (E in L)  [transition]
```

The rule says that `SetEvent(I,E);` is the next API to be executed by a running task, where `I` is a task ID and `E` is an event name. Task `I` is in the *waiting* state, and is waiting for a list `L` of events. If `E` is in the list, task is transferred to *ready* state, and it does not wait for any events. Thus, the list `L` is changed into an empty one, represented by `L => .` in the cell waitEvents. The status of event `E` is changed into `SET` in the cell taskEvents. Task I is added to the tail of the sub-list of ready tasks which have the same priority as `I` in the list of ready tasks by function `add2Tail`. At the same time, the `schedule` signal is fired to invoke the scheduler.

If task `I` is not waiting for the event `E`, the event is simply set after the API is called. We omit the formal definition of it and those of other three event-related APIs in the paper.

5.5 Formalization of Error Handling

There are pre-defined errors in the OSEK/VDX standard. Such errors should be handled correctly when an operating system is implemented. For instance, an error will occur when a task tries to terminate itself while occupying some resources, which is strictly forbidden. If an error is raised, a specific error code should be returned. However, the standard does not specify how to handle such errors, and it is up to system developers.

We formalize errors by a specific function called `Error`, which takes four arguments, i.e., an error code, the identifier of the task which causes the error,

the API that causes the error, and a string to provide detailed information of the error. When an error occurs, an error cell errorCell as shown in Fig. 5 is created with a term constructed by the `Error` function in it. At the same time, an error signal is inspired and put in the signal cell. The following rule shows an example of the formalization of the error which is caused when a task tries to terminate while still occupying resources.

```
rule <task> <state> running </state> <k> TerminateTask(); </k>
     <tid> I </tid> <accRes> ListItem(R) ... </accRes> ... </task>
     <signal> . => stop </signal>
     (.Bag => <errorCell> Error(E_OS_RESOURCE, I, TerminateTask();,
     "Task cannot terminate when occupying resources!") </errorCell>)
[transition]
```

The term in cell accRes is the list of resources that are currently occupied by the task. In the above rule the list is not empty when the API `TerminateTask();` is to be executed in the next step. In the cell errorCell, `E_OS_RESOURCE` is an error code which is pre-defined in the standard.

5.6 Formalization of OIL

We formalize OIL in $\mathbb{K}$ in order to support user-defined configurations. $\mathbb{K}$ is well suited to formalize programming languages. OIL can also be naturally formalized in $\mathbb{K}$ like other languages such as C. One difference is that the semantics of OIL is formalized as structural rules, instead of computational rules. That is because OIL is a configuration language, which is used for configuring resources, tasks, events, etc. in a system, but not for computation or execution. Thus, we declare a set of structural rules which are used to construct an initial $\mathbb{K}$ configuration according to an input OIL program.

Given an OIL program, a $\mathbb{K}$ configuration is instantiated based on the declarations of resources, tasks, and events in the program. For instance, for each resource which is declared as shown in Fig. 4, a resource cell is created, which consists of four unit cells for resource identifier, its ceiling priority (0 at initial), the resource property, and the list of tasks that can access it (empty at initial). The following rule specifies the creation of a resource cell according to a declaration of a resource. The condition means that `I` is not used as an identifier for other tasks, events and resources.

```
rule <k> (RESOURCE I:Id { RESOURCEPROPERTY = RP; } ; => .) ... </k>
    <resources>(. => <resource> <rid> I </rid> <rPriority> 0 </rPriority>
       <rProperty> RP </rProperty> <rOwner> .List </rOwner> </resource>)
       ... </resources> <types> M:Map => (I |-> resource) M </types>
    <signal> . </signal> when notBool $hasMapping(M,I) [structural]
```

The ceiling priority of the resource and the list of tasks are calculated when tasks are initialized. If a task is declared to own a resource as shown in Fig. 4, it is added to the list. If the current ceiling priority (initially 0) is less than the priority of the task, it is raised to the priority of the task. The corresponding structural rules can be defined likewise. We omit the details in the paper.

6 Applications of the Formal Semantics

In this section, we demonstrate two applications of the formal semantics of the OSEK/VDX standard, i.e., to verify the properties of OSEK/VDX applications by model checking, and to generate test cases for the testing of OSEK/VDX operating systems.

6.1 Verification of OSEK/VDX Applications by Model Checking

There may be multiple tasks running in an OSEK/VDX application. It is necessary to verify that tasks can correctly synchronize and deadlock should never happen. Suppose that there are only two tasks t_1, t_2 in an application and two events e_1, e_2. Task t_1 is waiting for e_1 in order to set event e_2, while t_2 is waiting for e_2 in order to set event e_1. Both the two tasks are in *waiting* state, leading to deadlock.

An OSEK/VDX application consists of two parts: one is application configuration which describes the basic information of resources, tasks, events, etc., in the application, and implementation of each task. Application configurations are defined by OIL, and tasks are implemented in some programming language such as C. Thus, it requires formalizing the semantics of a specific programming language in which tasks are implemented, as shown in Fig. 2. As mentioned in Sect. 4, $\mathbb{K}$ can be easily used to define the operational semantics of programming languages. In this paper, we use a simple C-like imperative programming language whose semantics has been formally defined in $\mathbb{K}$ [13], to demonstrate the feasibility of verifying OSEK/VDX-based applications. We integrate the semantics of the language and the semantics of the OSEK/VDX standard. With the integrated semantics, we verify the OSEK/VDX applications that are implemented in the simplified language.

Figure 6 shows a simplified application which is used to monitor tire pressure [14]. There are four tasks with different priorities. Task `MT` is used to repeatedly activate task `RT` (line 34), which collects data from tire sensor and then activates task `ST`. Task `ST` puts the collected data into buffer (line 47) and activate task `PT` to process the data (line 49). The synchronization between task `RT` and `ST` is achieved by an event `evt`. Task `ST` has to wait for the event until the event is set by `RT` (line 45, 41).

The application is supposed to run repeatedly. We verify if the application is free of deadlock by searching all possible execution results of the application using the formal semantics of the standard and the simplified programming language. The command is as follows:

```
krun tire-app.osek --search-final
```

`krun` is a $\mathbb{K}$ command, used to run a program with the formal semantics which is predefined in $\mathbb{K}$ for the language in which the program is implemented. In this example, the program is saved as a file named `tire-app.osek`. The option `search-final` means that `krun` will return all possible final states of the program. With the command, `krun` returns a final state, which means that the

```
1  EVENT evt {
2     MASK = AUTO;
3  };
4  RESOURCE BUFF{
5    RESOURCEPROPERTY = STANDARD;
6  };
7  TASK MT {
8    PRIORITY = 4;
9    AUTOSTART = true;
10   SCHEDULE = FULL;
11 };
12 TASK RT{
13   PRIORITY = 6;
14   AUTOSTART = false;
15   SCHEDULE = FULL;
16 };
17 TASK ST{
18   PRIORITY = 8;
19   AUTOSTART = false;
20   SCHEDULE = FULL;
21   EVENT = evt;
22   RESOURCE = BUFF;
23 };
24 TASK PT{
25   PRIORITY = 10;
26   AUTOSTART = false;
27   SCHEDULE = FULL;
28   RESOURCE = BUFF;
29 };
```

a. Configuration

```
30 int data;
31 int buff;
32
33 TASK MT{
34   while(true){ActivateTask(RT);}
35 };
36 //Terminate
37 TASK RT{
38   //get data from tire sensor
39   ActivateTask(ST);
40   if(data!=0)
41     {SetEvent(ST,evt);   }
42   TerminateTask();
43 };
44 TASK ST{
45   WaitEvent(evt);
46   GetResource(BUFF);
47   buff=data;
48   ReleaseResource(BUFF);
49   ChainTask(PT);
50 };
51 TASK PT{
52   //Read data from buff
53   GetResource(BUFF);
54   // process data
55   buff=0;
56   ReleaseResource(BUFF);
57   TerminateTask();
58 };
```

b. Implementation of tasks

Fig. 6. The configuration and source code of an OSEK/VDX application

program cannot proceed from that state. The returned state shows that all the four tasks cannot be executed. In the state, task `RT` is in *running* state, and the statement it executes next is `ActivateTask(ST)` (line 39). However, task `ST` is in *waiting* state. According to the OSEK/VDX standard, there is a maximum number of task activation. If the number of activation times exceeds the maximum number, an error occurs. In our experiment, we assume the maximal number is 1. In this case, an error occurs because it violates the maximum activation count.

We found the reason why the *running* task called the *waiting* task in the application by checking the execution path from the initial state to the returned state. 4 The execution path shows that after `RT` activates `ST` (line 39), `ST` starts to run because it has a higher priority than `RT`. However, `ST` goes to *waiting* state because `evt` is not set (line 45). The scheduler selects `RT` to run because among the ready tasks, i.e., `RT` and `MT`, `RT`'s priority is the highest. However, `RT` does not set the event because `evt data` is 0 (line 40). It just terminates itself (line 42). `MT` is the only ready task, which is selected to run by the scheduler. It activates `RT` (line 34), and `RT` tries to activate `ST` (line 39). However, `ST` is in the *waiting* state, leading to the deadlock. The problem is caused by the code at line 40 and 41 because `evt` cannot be always set after `ST` is activated. We can fix it by moving `ActivateTask(ST);` to the block of `if` condition. We execute the revised application by *searching* with the same command. No solution is found, which

means that there is no deadlock state from which the system cannot proceed further. Namely, the program is free of deadlock.

OSEK/VDX-based applications are implemented usually in C. The semantics of C has been formalized in $\mathbb{K}$ [7]. We believe that by integrating the semantics of C and the standard we can verify more complicated OSEK/VDX-based applications implemented in C, which is one piece of our future work.

6.2 Using the Formal Semantics for Test Case Generation

In this section, we show that the formal semantics of the standard in $\mathbb{K}$ can be used to generate test cases for the conformance checking of OSEK/VDX-based operating systems. Testing is still the main approach to conformance checking of OSEK/VDX-based operating systems. For example, an automobile operating system must pass a set suite distributed by a certification agency in order to get a certificate for the compliance of the system with the OSEK/VDX standard. However, it is practically impossible to test all possible combinations of APIs. One solution is to analyze the constraints among APIs and to generate automatically test cases that satisfy these constraints [4,6]. Given some constraints and a configuration of tasks, resources, and events, we would like to generate a sequence of APIs for each task and the generated sequence of APIs satisfies the specified constraints. Specifically, we provide an initial state where an OSEK/VDX application is configured, i.e., events, resources and tasks are defined. Tasks are not implemented, that is, there are no statements defined for tasks. We specify the target states which the operating system can reach from the given initial state by executing the application with a sequence of APIs for each task. Test case generation problem is to generate such API sequence for each task.

The formal semantics of the OSEK/VDX standard in $\mathbb{K}$ can also be used for generating such test cases. The basic idea is as follows. The input is the configuration part of an OSEK/VDX application. After the configuration is loaded by the $\mathbb{K}$ tool, a task cell is instantiated for each task according to their corresponding setting in the configuration and the task is in the *suspended* state. Only the task whose `AUTOSTART` property is true becomes ready and then is scheduled to run. In the initial state, all the tasks do not have any API to execute. We define a set of $\mathbb{K}$ rules which randomly generate an API for the currently running task based on the state of the task. The generated API is then executed based on its formal semantics that is predefined, and the state of the running task is changed correspondingly.

Generated APIs must satisfy some built-in constraints. These constraints are also specified in $\mathbb{K}$ rules. For instance, we define a rule which generates the API `ReleaseResource` with a resource `R` where `R` is predefined in the configuration of an application. Namely, the statement to be generated is `ReleaseResource(R);`. For the task that is go to execute the generated statement, the resource `R` must be the latest one that is allocated to the task, i.e., the resource `R` must be at the head of the list of resources in the cell taskResources. The corresponding $\mathbb{K}$ rule is defined as follows:

```
rule <task> <tid> I </tid> <state> running </state> <apiData> . </apiData>
  <k> . => ReleaseResource(R); </k> <taskRes> ListItem(R) ... </taskRes>
  ... </task> <tcgMode> true </tcgMode> <signal> . </signal>  [transition]
```

The rule also represents three conditions when API can be generated, i.e., the system is running in the test case generation mode as indicated by the cell tcgMode, the running task is undefined as indicated by an empty cell apiData, and no signal is waiting for handling as indicated by the empty cell signal.

If the randomly generated API is `TerminateTask` or `ChainTask`, a sequence of APIs are completed for the running task. That is because there is a constraint that `TerminateTask` and `ChainTask` must be the last API in a task. After the API is executed, the task is terminated and the scheduler selects another task to run according to the scheduling policy. After each task has a generated sequence of APIs, a test case is generated.

As an example, we explain how to generate test cases with the configuration defined in Fig. 6. Each test case consists of four sequences of APIs for the tasks in the configuration. We feed the configuration into the $\mathbb{K}$ tool and a pattern to which expected results should match. The pattern specifies the constraints to the target states such as the number of APIs for each task. There are two optional parameters specifying the maximal searching depth and the number of test cases. The following command is an example used for test case generation.

```
krun tire-conf.osek -c TCG=true --search --pattern="<task> <historyK> I1:
    ListItem I2:ListItem </historyK> <done> true </done> BG1:Bag </task>
   <task> <historyK> J1:ListItem J2:ListItem </historyK> <done> true </done>
   BG2:Bag </task> <task> <done> true </done> BG3:Bag </task> <task> <done>
   true </done> BG4:Bag </task>" --depth=30 --bound=100
```

It takes the configuration of the tire monitor application as input. `TCG` is an argument which is used to initialize the cell `tcgMode` as true so that the rules defined for test case generation can be executed. The pattern in the command specifies that in the target states there are at least two tasks which have exactly two APIs. The last two optional arguments means that the maximal searching depth is 30 and the bound of solutions is 100. By the above command, the $\mathbb{K}$ tool returns a set of $\mathbb{K}$ configurations that match the specified pattern. In each configuration, every task is assigned with a sequence of APIs. All the tasks with assigned sequences of APIs constitute a test case.

Table 1. Experimental result of generating two classes of test cases with pattern A (the left table) and B (the right table).

Depth	Solution	Test case	Time (sec)
16	0	0	6
17	92	23	10
20	1024	132	30
21	1468	163	43
22	1748*	200*	65

Depth	Solution	Test cases	Time (sec)
20	0	0	16
21	176	44	23
25	1320	186	76
26	1572	209	97
27	1736*	234*	146

The number of test cases may be infinite, leading to the non-termination of generation. For instance, a task can infinitely repeat the process of getting and releasing a resource, making the process of test case generation non-terminating. We can solve this problem by setting upper bounds to the number of generated test cases and the number of APIs in each task respectively. In the experiment, we defined two patterns denoted by A and B, specifying that in each generated test case there must be at least two tasks which have exactly two and three APIs, respectively. The experimental result is shown in Table 1. Solutions are the configurations returned by $\mathbb{K}$, and they match the specified pattern. We extract the generated APIs in cell apiData from the configurations returned by the $\mathbb{K}$ tool, and obtain a number of test cases. The numbers with $*$ mean that they are the upper bound at the corresponding depth. The $\mathbb{K}$ tool runs out of memory once the upper bound exceeds that numbers.

7 Related Work

There is some formalization in different formal languages of the OSEK/VDX standard. In [15], the semantics of APIs in the standard is formalized using Hoare-logic. The purpose is to verify the correctness of the APIs that are implemented in concrete OSEK/VDX-based operating systems, which is different from ours. The semantics of the standard is formalized using Promela in [6] and NuSMV in [4], with the purpose of test case generation. Compared with their formalization, our formal semantics in $\mathbb{K}$ is more flexible and generic in that they have to assume that the list of ready tasks in system is finite because the languages requires the system specified must be of finite-state, while in our formalization we do not have that restriction. Their formalization also requires extra effort to be instantiated according to concrete user-defined applications, while our formalization directly accepts user-defined applications as input without any transformation. Their approaches to test case generation are different from the one described in this paper. For instance, in [4] they use automata to control what is the next possible API to call based on user-given constraints. This can improve the efficiency by avoiding unnecessary trial of those APIs that violate the constraints if they are called. We have tried implementing the automata-based approach in Maude and evaluated that Maude can be used as an efficient test case generator [3]. Since Maude is the underlying rewrite engine of $\mathbb{K}$, we believe that this approach can also be implemented in $\mathbb{K}$ based on our formal semantics of the OSEK/VDX standard. As a piece of future work, we also consider integrating the automata-based approach into our formalization to improve the efficiency of test case generation.

The OSEK/VDX standard is also formalized in CafeOBJ [16] with the purpose of defining a formal specification of the standard so that we can verify the conformance of an automobile operating system to the standard by verifying whether it conforms to the formal standard. However, it is still a challenging problem to check the formal specification contains no contradiction as the authors mentioned [16]. The standard is also formalized using Event-B in [17].

However, in their work they do not explain how to use their formalization for verification. The common problem with the two existing formalization is how they can be used for formal verification. Our work shows that the formalization of the standard in $\mathbb{K}$ can be effectively used for the verification of concrete OSEK/VDX-based applications and operating systems.

8 Conclusion and Future Work

We have presented a formal semantics of the OSEK/VDX standard, which is defined in the $\mathbb{K}$ framework. We demonstrated two applications of using the formal semantics, i.e., verification of OSEK/VDX-based applications by model checking and generation of test cases for the testing of OSEK/VDX-based operating system. Compared with the existing formalization of the standard, the formal semantics in $\mathbb{K}$ is more flexible and generic in that there is no restriction to the number of tasks, resources and events in the formalization, and it does not need extra effort to instantiate the formal semantics with user-defined applications. Another advantage of the formal semantics is that it can be integrated with the semantics of other prevalent programming languages such as C in order to verify OSEK/VDX-based applications which are implemented in those languages. This work also shows that $\mathbb{K}$ is not only a semantics framework for the definition and formalization of programming languages, but for automobile operating systems such as OSEK/VDX-based operating systems.

In our current formalization, we do not consider some functionality in the standard such as interruption and real-time feature, which are also equally important to task scheduling, etc. As one piece of our future work, we will formalize them based on the current work and develop a tool for the formal analysis of OSEK/VDX applications with interruption and real-time features. Another piece of future work is to improve the efficiency of test case generation by integrating the automata-based approach proposed in [3] into our formalization and to support user-defined constraints for test case generation by formalizing the OSEK constraints specification language defined in [4]. We also consider integrating the formal semantics of C in $\mathbb{K}$ with the formal semantics of the standard for the verification of more complicated OSEK/VDX-based applications that are developed in C.

References

1. OSEK Group, et al.: OSEK/VDX Operating System Specification (2009)
2. John, D.: OSEK/VDX conformance testing-MODISTARC. In: Proceedings of OSEK/VDX Open Systems in Automotive Networks, IET (1998)
3. Choi, Y., Zhang, M., Ogata, K.: Evaluation of Maude as a test generation engine for automotive operating systems, pp. 1–15 (2013) (Manuscript)
4. Choi, Yunja: Constraint specification and test generation for OSEK/VDX-based operating systems. In: Hierons, Robert M., Merayo, Mercedes G., Bravetti, Mario (eds.) SEFM 2013. LNCS, vol. 8137, pp. 305–319. Springer, Heidelberg (2013)

5. Roşu, G., Şerbănută, T.F.: An overview of the $\mathbb{K}$ semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010)
6. Yatake, Kenro, Aoki, Toshiaki: Automatic generation of model checking scripts based on environment modeling. In: van de Pol, Jaco, Weber, Michael (eds.) Model Checking Software. LNCS, vol. 6349, pp. 58–75. Springer, Heidelberg (2010)
7. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th POPL, pp. 533–544. ACM (2012)
8. Guth, D.: A formal semantics of Python 3.3, Master thesis (2013)
9. Meredith, P., Hills, M., Roşu, G.: An executable rewriting logic semantics of $\mathbb{K}$-Scheme. In: Workshop on Scheme and Functional Programming, vol. 1 (2007)
10. Zahir, A.: OIL-OSEK implementation language. In: OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEE Seminar, IET, pp. 1–8 (1998)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7–8), 721–781 (2012)
13. Roşu, G., Şerbănută, T.F.: $\mathbb{K}$ overview and simple case study. In: Proceedings of International K Workshop (K'11), ENTCS. Elsevier (2013) (to appear)
14. Zhang, H., Aoki, T., Yatake, K., Zhang, M., Lin, H.H.: An approach for checking OSEK/VDX applications. In: Proceedings of the 13th QSIC, IEEE CSP, pp. 113–116 (2013)
15. Huang, Y., Zhao, Y., Zhu, L., Li, Q., Zhu, H., Shi, J.: Modeling and verifying the code-level osek/vdx operating system with csp. In: Proceedings of the 5th TASE, IEEE CSP, pp. 142–149 (2011)
16. Yatsu, H., Ando, T., Kong, W., Hisazumi, K., Fukuda, A., Aoki, T., Futatsugi, K.: Towards formal description of standards for automotive operating systems. In: Proceedings of 6th ICSTW, IEEE CSP, pp. 13–14 (2013)
17. Vu, D.H., Aoki, T.: Faithfully formalizing OSEK/VDX operating system specification. In: Proceedings of the 3rd SoICT, pp. 13–20. ACM (2012)