2016 10th International Symposium on Theoretical Aspects of Software Engineering

Verifying OSEK/VDX OS Design using Its Formal Specification

Dieu-Huong Vu*, Yuki Chiba†, Kenro Yatake†, Toshiaki Aoki†
*Vietnam National University Hanoi, Vietnam
† Japan Advanced Institute of Science and Technology, Japan

***Abstract*—Automotive systems are widely used in industry and our daily life. As the reliability of automotive systems is becoming a greater challenge in our community, increasingly more automotive companies are interested in applying formal methods to improve the reliability of automotive systems. We focus on automotive operating systems conforming to the OSEK/VDX standard. Such operating systems are considered as important components to ensure the reliability of the automotive systems. In previous work, we proposed a framework to verify the design models of reactive systems against their specifications. This framework allows us to check whether the design model conforms to the specification based on a simulation relation. This paper shows a case study in which the framework is applied to a real design of the OSEK/VDX operating system. As a result, we found that we were able to check several important properties of the design model. We show the effectiveness and practicality of the framework based on the results of the case study.**

Keywords*—OSEK/VDX OS, formal specification, design model, formal verification, model checking, simulation relation.

I. Introduction

As an automotive industry standard of operating system specification, OSEK/VDX OS specification [12] is widely applied in the process of designing and implementing the operating system (OS) for automotive systems. In order to obtain a high-reliability OS, a design model is often developed in advance, and some verification techniques like model checking [3] are employed to check whether the design conforms to the OSEK/VDX OS specification. If the design model has been ensured then the prototype can be developed following the design model.

We are working on a design of an OS compliant with the OSEK/VDX standard. The aim of this work is to provide a high quality OS by applying automated formal verification. To conduct the verification process completely, model checking as an exhaustive technique can be used to check the design model with given properties. This approach is adopted in [4] where the design is described in Promela/Spin [6] and the properties of interest are specified as temporal logic formulas [14]. The advantage of using Promela is that it allows us to design the highly optimized behavior of the OS in an imperative manner using various data structures. However, we consider that temporal logic formulae, which allows us to describe properties about invariants on some variables and the relative order of event calls, are not adequate for describing the important properties of the OS. What we need to verify about the OS is the correctness of the scheduling which can be precisely described by specifying the pre-condition and the post-condition of each event. For example, when an activation event of a task is called, the task must become running and the currently running task must become ready in the states just after the event is called. To specify such a property in temporal logic, as discussed in [5], we need to explicitly define the execution steps of the events. This makes the formulae complex and prone to mistakes. Whereas, by using the rich notions (e.g. sets and relations) in the formal specification languages like Event-B[1][13], one could easily describe such properties. Therefore, we intentionally use Event-B to facilitate describing properties of the OS. In Event-B, one can describe the system as a set of events and the behavior of each event can be specified as pre-conditions and post-conditions using the rich notions. It also provides a facility to verify the consistency and the correctness of the properties. For these reasons, Event-B and Promela are intended to describe the specification and the design in our verification of the OS design. Accordingly, we verify the conformance of the Promela design to the Event-B specification. In such a combination between Event-B and Promela/Spin, the OS design could be verified with respect to a reliable specification.

In previous work, we proposed a framework to verify the design models of the reactive systems against their specifications [16]. The framework includes three main steps. Firstly, a labeled transition system (LTS) is generated from the specification. Next, from each state appearing in the LTS, verification conditions which must be met by the corresponding state of the design are generated. Finally, the design in combination with the LTS is input into a model checker to check the verification conditions. In this way, one can check the correspondence of state transitions, or the simulation relation, between the specification and the design. This shows that the design conforms to the specification.

In this paper, we present a case study of applying the framework to the verification of a real OS design compliant with OSEK/VDX standard. For applying the framework, on the one hand, we formalize the OSEK/VDX OS specification in Event-B and make sure the consistency of the specification before using it to verify the design [15]. On the other hand, we determine the reasonable bounds for the verification. As we mentioned earlier, when checking the simulation relation between the specification and the design, we need to generate the LTS from the specification. The problem is that generating all possible execution sequences from the specification makes the size of the LTS so large that it has a tendency to cause the state explosion when we apply model checking. To avoid this, we apply the existing framework flexibly: various ranges are considered to give appropriate bounds for the verification; and they need to be defined depending on the properties to be checked. In the case study, we mainly explain how we defined the appropriate bounds to check our desired properties. We also show the results of the experiments and evaluate the effectiveness and the applicability of the framework in a practical setting.

This paper is organized as follows. In section II, we present the formalization of the OSEK/VDX OS specification in Event-B. In section III, we present our verification target, a real design of the OS. In section IV, we present a workflow to apply the framework in verification of the OS design using its specification in Event-B. In section V, we present verification results of various properties. In section VI, we discuss the effectiveness of the framework. In sections VII and VIII, we present related works and conclusions.

II. Specification of OS in Event-B

In this section, we present formalization of the OSEK/VDX OS specification in Event-B.

OSEK/VDX OS Specification. The specification mainly describes entities managed by the OS (e.g., tasks, resources, interrupt routines) and services of the OS for controlling automotive systems. Tasks are the basic building blocks of an application program. They transit between several states (e.g., suspended, ready, waiting and running). The OS makes task state-transitions whenever necessary using some system services. The OS provides the following service groups: task management, resource management, event control, and interrupt management. The OS services are called either by tasks or internally within the OS. In the task management, the OS's scheduler decides the execution of tasks based on priorities and the task activation order. It uses multiple ready queues to store instances of tasks that are currently in the ready state. Tasks with the same priority are stored in the same ready queue according to the task activation order. The scheduler always chooses one task with the highest priority among those ready tasks. Within the set of tasks in the ready state and of highest priority, the scheduler finds the oldest task to be executed.

Formalizing OSEK/VDX OS Specification. In order to formalize the specification in Event-B, we identify essential behaviors that must be verified. We focus on not only individual services of the OS but also combination of these services. Therefore, we formalize the services of interest in Event-B including regular and irregular aspects. Formal specification in Event-B mainly consists of state variables, operations (events) on the variables, and state invariants. Figure 1 illustrates the structure of the specification in Event-B. The variables such as `tasks`, `res` represent all the created tasks and the managed hardware resources. The invariants represent constraints, e.g. at any time only one task is in running state. System services are formally defined as *guarded events* like `ActivateTask` with guard conditions, e.g. task `t` is in a suspended state, and actions that make the state transitions, e.g. transferring `t` to ready state and pushing it into the corresponding ready queue. This is a highly abstracted level description of the OS; it specifies result of operations rather than details of how to make the results.

As a result, the specification in Event-B contains a list of state variables, a list of events which modify states (or state variables), and a list of invariants which are preserved by events (or transitions). Thus, the possible execution sequences of such specification can be represented as an LTS.

```
VARIABLES
        tasks, res, evt, isr, tpri, tstate, rdyQuItem, qsize, rdyQSet, acnt
INVARIANTS
        tasks⊆TASK, rdyQSet⊆tasks, acnt∈ℕ1
        qsize ∈ 0‥MAXPRI → ℕ1
        rdyQuItem ∈ 0‥MAXPRI × 0‥MAXQSIZE → rdyQSet
        tstate ∈ tasks → STATE
        ∀ta,tb·ta∈tasks∧tb∈tasks∧tstate(ta)=run∧tstate(tb)=run⇒ta=tb

EVENTS
ActivateTask  ≙
        any t where t∈tasks, tstate(t)=sus, acnt(t)<MAXACT
        then tstate:|tstate'∈tasks→STATE ∧ (tstate(t)=sus ⇒ tstate'(t)=rdy)
                rdyQSet:|rdyQSet'⊆ tasks ∧ rdyQSet' = rdyQSet ∪ {t}
                rdyQuItem(tpri(t)↦qsize(tpri(t))+1)≔t
                qsize(tpri(t))≔qsize(tpri(t))+1
```

Fig. 1: Formal Specification in Event-B

III. OS Design Model in Promela

The OS design defines a collection of functions which realize the services in the specification. The OS design is more concrete than the specification: the specification describes desirable results of the services that the OS provides, whereas the design additionally includes the details of how to make the results. In order to make sure the conformance, we should verify whether the results provided by the functions in the design are the same as the results described in the specification.

The OS is an open system. It does scheduling of the tasks if it gets stimulus such as system call invocations from its environment. The environment of the OS includes applications running on the OS and hardware which causes interrupts. As mentioned earlier, the OS design only defines a collection of functions, it cannot operate by itself. To operate it, we need an environment which calls functions of the OS; the design must be verified in communication with the environment. As we explained later, the environment is constructed from the specification, and input to Spin to check the simulation relation between the specification and the design.

```
typedef TCB {int id, pr, dpr, … }
typedef RCB {int id, pr, tid, …  }
TCB tsk[5];
RCB res[5];
int ready[25];
TID turn;
inline schedule() { … }
inline enq(pr, id, q) { … }
inline  _DeclareTask(id, pr) { … }
inline  _ActivateTask(id) { … }
inline  _ChainTask(id, tid) { … }
inline  _TerminateTask(id) { … }
```

```
inline _ActivateTask(id){

if
:: tsk_state[ret_ix].actcnt <OS_ACT_MAX
   -> tsk_state[ret_ix].actcnt++;
          if
          :: tsk_state[ret_ix].tstat==SUSPENDED
             -> enq_prio(tsk_state[ret_ix].tpriority,id, q);
                  tsk_state[ret_ix].tstat = READY;
…}
```

Fig. 2: OS Design in Promela

Promela allows us to describe the design in an imperative manner. Functions of the OS can be described by using inline functions. Figure 2 (left) illustrates the whole structure of the OS design. We call this model a *design model*. We constructed the OS design according to the approach proposed by [2]. Our OS design is described in about 2800 lines of Promela code. It first defines implementable data structures such as `tsk`, `res`, and `ready` which represent an array of tasks, an array of resources, and ready queues, respectively. Following these data structures, a set of functions are defined. For example, `_ActivateTask` and `_TerminateTask` are the functions to perform activation and termination of tasks, respectively. Figure 2 (right) illustrates the body of the functions where design decisions to realize the behaviors are explicitly described using various control structures. It also consists of assignment statements over the variables. The variables such as `tsk`, `res`, and `ready` represent information about the system (states). The execution of statements changes the values of variables. Therefore, the model in Promela can be interpreted as an LTS if we consider that the variables are states and each function call is a label to make transitions on the states.

IV. CHECKING OS DESIGN MODEL

According to the proposed framework, verifying a design against its formal specification is based on a simulation relation between them. Here, the Event-B specification and the OS design are interpreted as LTSs. We now present the simulation relation between two LTSs and present a workflow to apply the framework in verification of the OS design using its formal specification in Event-B.

Suppose that M1 and M2 be two LTSs. We define M2 simulating M1 based on semantics of LTSs by extending the given relation on the states. The states are value assignments which are mappings from the variables to the values. Therefore, the relation on states of M1 and those of M2 are established based on mappings R and C where R is the mapping from variables of M1 to those in M2, C is the mapping from values in M1 to those in M2. Figure 3 (left) shows a relation between state p of M1 and state q of M2. p relates to q based on R and C because $u = sus$ in state p corresponds to $v = 1$ in state q with mappings $R(u) = v$ and $C(sus) = 1$. M2 simulates M1 if for each transition in M1 from state p to state p' and p relates to state q of M2, there exists state q' and a corresponding transition in M2 from q to q' such that p' relates to q'. In Figure 3 (right), a line arrow connecting p to p' represents a one-step transition from p to p', and a dashed arrow connecting q to q' represents an n-step transition from q to q'. To check whether M2 simulates M1, we check whether there exists a reachable state q' from q such that $v = 2$ corresponding to $u = rdy$ in p' with mappings $R(u) = v$ and $C(rdy) = 2$. We refer the readers to [16] for a more formal definition of the simulation relation.

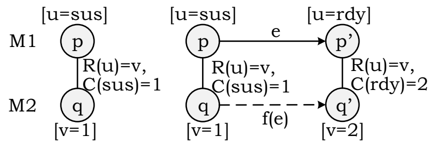


Fig. 3: Simulation Relation

The overall workflow is shown in Figure 4. Firstly, we formalize the OSEK/VDX OS specification in Event-B. The consistency of the behaviors and the properties is ensured in this step. Secondly, to avoid the state explosion, we give reasonable bounds for the verification. In this step, we pick up behavior scenarios of the OS from the OSEK/VDX OS specification according to the properties we want to verify. The scenarios provide examples of the system behaviors which satisfy the desirable properties. Based on the behavior scenarios, we determine bounds for the verification so that when we apply them to the Event-B specification, the execution sequences of the specification within the bounds cover at least the behaviors under consideration and prevent the state explosion. Finally, we apply the proposed framework to check whether the OS design conforms to the Event-B specification within the bounds. This process includes: generating the LTS of the specification; translating it into Promela to generate the environment and assertions using mappings between elements in the specification and those in the design; and applying model checking to verify the design in communication with the environment against the assertions. Here, the environment is to trigger functions in the design because the design of reactive systems does not execute by itself; and the assertions represent verification conditions - constraints on the simulation relation between the design and the specification. After the verification has been completed, if no error is returned, the OS design conforms to the specification: the results provided by the functions in the design are the same as the results described in the specification via the mappings. Otherwise, a transition in the specification is not followed by the design. In the latter case, we can show that an error really exists in the design.

Defining Bounds. Model checking does an exhaustive check of the system. It needs a representation of the system as a finite set of all possible states. Firstly, abstract types in Event-B, e.g. `TASK` in Figure 1, must be replaced by concrete types. Also, types having infinite ranges of values like Int and Nat must be restricted as finite ranges. Then, by studying the properties of interest, we can restrict the behaviors will be checked. Such restrictions are to reduce the size of LTS explored from the Event-B specification. We define such restrictions as bounds of the verification.

The size of the LTS depends on ranges, denoted V, E, and D, where V is range of values for every variable; E is range of operating system services which are defined as guarded events in Event-B; and D is the depth of the execution of the Event-B specification. In order to restrict V, we define the finite domain replacing for the infinite domain. Within this restriction, the state space and the set of transitions of the LTS become finite sets. However, the size of the LTS could be so large that it causes the state explosion. Therefore, we may need additionally restrict E or D to reduce the size of the LTS. To restrict E, we define restricted sets of system services relevant to intended properties. In this way, we separate verification to deal with distinct groups of system services. As a result, set of events that may be enabled in states is reduced; thus, set of transitions that may be triggered in states of the LTS is also reduced. For D, we give a value for maximum depth of execution sequences explored from the specification. This is useful to check the properties among different groups of system services while avoiding the state explosion.

We determine bounds by studying the properties and the behavior scenarios of the target system. This step is explained in Section V.

Generating an LTS from specification. In order to generate the LTS from the specification and bounds, our LTS Generator computes all possible transitions and reachable states. Every value used in the computation must be within the bounds. Starting at the initialization, the generator enumerates all possible values for the constants and variables of the specification that satisfy the initialization and the invariant to compute the set of

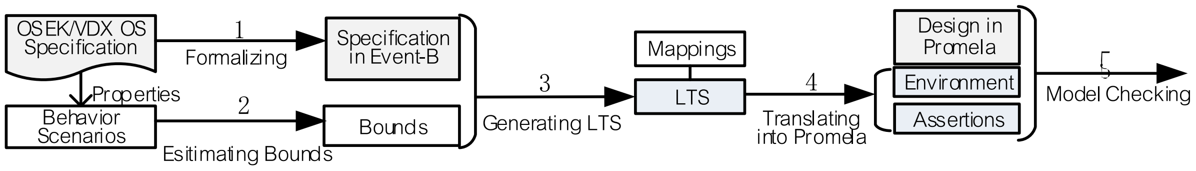


Fig. 4: Checking Design using Its Formal Specification (Workflow)

initial states. To compute all possible transitions from a state, the generator finds all possible values for event parameters of an individual event to evaluate the guard of that event. If the guard holds in the given state, the generator computes the effect of the event based on substitution of that event. When new states are generated, we repeat this process to these states until no new state is generated.

Generating Environment. In the workflow, the OS design is verified in communication with its environment. The environment includes function calls; it triggers the specific behaviors of the designs by calling functions of the designs. We construct such comprehensive environments that they represent all possible behaviors described in the specification of the OS. In the previous step, we generated the LTS of the specification. In this step, we generate the environment by translating the LTS into Promela such that the enabled events in LTS are translated to the corresponding function calls in Promela. This is performed by our Promela Code Generator.

Figure 5(left) demonstrates an LTS, which is generated from the specification of the OS. The LTS represents possible sequences of state transitions within the bounds. Here, the rectangles represent the states and the labeled arrows represent the events that are enabled in each state. For example, two events `AT(t1)`, `AT(t2)` are enabled in state $s0$. In the framework, the states are defined as the value assignments; however, we show them here as values, e.g. `(sus, sus, sus)`, for readability. The LTS is translated into Promela to generate the environment (from left to the right of Figure 5). For this generation, we give a mapping from the events in the LTS to the function calls in the environment. It could be one-to-one or many-to-one mapping. Figure 5 shows a sample case of one-to-one mapping. Here, event `AT(t1)` in the LTS is mapped to function call `_ActivateTask(task1.tid)` in the environment; also, event `TT(t1)` is mapped to function call `_TerminateTask(task1.tid)`. The states and transitions in the LTS are represented by labels and if-statements in the environment. By combining the design model and the environment model, we obtain the combination model, which will be input to the model checker in the last step of the workflow.

Generating Assertions. Verification conditions, which represent constraints on the simulation relation between the specification and the design, are encoded as assertions. They will be checked by Spin. From each reachable state of the LTS, we generate an assertion that must be met by the corresponding state of the design. This generation is based on the mappings R and C from the variables, the values in the specification to those in the design. This is also performed by our Promela Code Generator. In sample case of Figure 3 (right), for example, from state p' where $u = rdy$ at the top with mappings $R(u) = v$ and $C(rdy) = 2$, the generator outputs an assertion $v = 2$ to check whether there exists corresponding state q' at the bottom.

In this workflow, we combine the consistency proofs in Event-B and the model checking with Promela/Spin to verify invariants of the specification and make sure that operations described in the design preserve the pre-conditions and the post-conditions. Not only individual function call but also the relation of function calls is also taken into account when we generate all possible sequences of function calls to construct the environment. This is important to confirm that every behavior is checked within the bounds.

V. Case study

Our target system is an operating system compliant with OSEK/VDX standard. Our case study was carried out with inputs including the OSEK/VDX OS specification, the Event-B specification, and the OS design described in Promela. We focus on the verification of properties concerned with the correctness of scheduling.

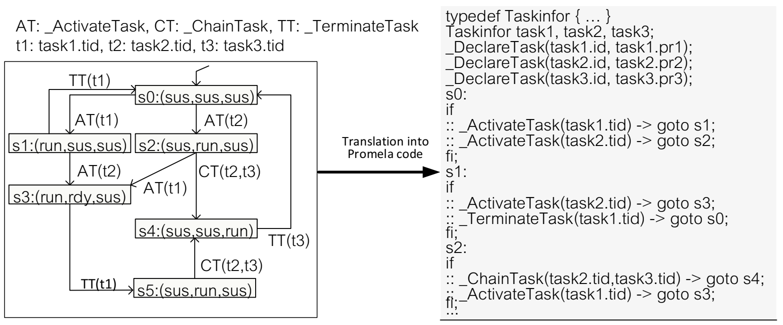
Properties and Bounds. Scheduling is concerned with entities such as tasks, ready queues, resources, events, and interruption routines. We show some properties of scheduling in Table I.

TABLE I: Properties

Prp.	Description
P1	A task with lower priority is preempted by a task with higher priority (full preemptive scheduling)
P2	A terminated or chained task goes into the suspended state
P3	An extended task in the waiting state is released to the ready state if at least one event for which the task is waiting for has occurred
P4	If a task or interrupt routine requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task is raised to the ceiling priority of the resource
P5	If a task or interrupt routine releases the resource, the priority of this task is reset to the priority which was dynamically assigned before requiring that resource
P6	The index value is within the bounds of the array
P7	A task must not terminate or chain another task while holding resources
P8	Over activation of a task is prohibited

Bounds are determined based on the properties and behavior scenarios of the target system. The scenarios provide examples of the intended system behaviors which satisfy the desirable properties. Each scenario represents a partial behavior of the system. In the following, we analyze the behaviors of full preemptive scheduling to illustrate how to determine appropriate bounds for verification of such behaviors.

Figure 6 visualizes a scenario which represents desirable behaviors of full preemptive scheduling. This scenario de-



```
typedef Taskinfor { ... }
Taskinfor task1, task2, task3;
_DeclareTask(task1.id, task1.pr1);
_DeclareTask(task2.id, task2.pr2);
_DeclareTask(task3.id, task3.pr3);
s0:
if
:: _ActivateTask(task1.tid) -> goto s1;
:: _ActivateTask(task2.tid) -> goto s2;
fi;
s1:
if
:: _ActivateTask(task2.tid) -> goto s3;
:: _TerminateTask(task1.tid) -> goto s0;
fi;
s2:
if
:: _ChainTask(task2.tid,task3.tid) -> goto s4;
:: _ActivateTask(task1.tid) -> goto s3;
fi;
...
```

Fig. 5: Generation of Environment from LTS

scribes the behaviors that satisfy property (P1). Here, the state transitions of two tasks, `T1` with priority `2` and `T2` with priority `1`, caused by `ActivateTask(T1)`. Initially, two tasks are both in the suspended state. Next, task `T2` transfers to running state after `ActivateTask(T2)` and task `T1` is still suspended. Then, `T2` activates `T1`. Due to the higher priority of task `T1`, task `T2` is preempted by task `T1`. By counting objects appearing in the scenario, we see 2 tasks, 2 different values for the task priorities, and 3 enabled events including `ActivateTask(T1)`, `ActivateTask(T2)`, and `TerminateTask (T1)` used to describe such behaviors. The least configuration of the OS to check such behavior includes 2 tasks and 2 different values for the task priorities. Therefore, the bounds applied to the Event-B specification for checking the behaviors under consideration are as follows: the ranges of values for `tasks` and `pri` are `{T1,T2}` and `{1,2}`, respectively; the range of operating system services includes `ActivateTask` and `TerminateTask`.

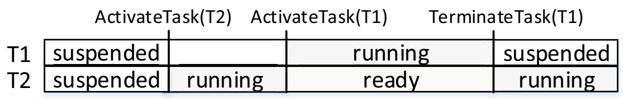


Fig. 6: Scenario representing P1

Property 6 is checked to make sure that the index value is within the bounds of the array. For example, as defined in the OS design, the bound of array `ready` is established by 72. To check whether the index value exceeds this bound. We can use 4 tasks, 2 multiple activation requests, and 10 values for the task priority. However, if we restrict only the range of values with 4 tasks, 2 multiple activation requests, and 10 values for the task priority, the verification could run out of memory, while we only need to call functions including declaration of tasks and activation of tasks to check such property. Therefore, an appropriate bound is to restrict both the range of values and the set of system services. Thus, $\widehat{\Sigma}_S$ includes `DeclareTask` and `ActivateTask`.

Table II shows bounds for checking the focused properties. Here, columns "Prp." lists the properties. As shown in Figure 1, variables `tasks`, `res`, `evt`, and `inr` define entities managed by the OS such as tasks, resources, events, and interrupt routines; variables `tpri`, `rpri`, and `ipri` define the priorities assigned to tasks, resources, and interrupt routines; and variable `tstate` defines the task state. Because of the space limitation, we show in column "V" restricted ranges of values for `tasks`, `tpri`, `res`, `rpri`, `evt`, `inr`, and `ipri` respectively. In column "E", we present the restricted set of OS services required for checking the corresponding properties, where DT, DR, DI, AT, CT, TT, GR, RR, WE, SE, SI, and RI stand for `DeclareTask`, `DeclareResource`, `DeclareISR`, `ActivateTask`, `ChainTask`, `TerminateTask`, `GetResource`, `ReleaseResource`, `WaitEvent`, `SetEvent`, `SetINTR`, and `ResetINTR`, respectively. Column "D" presents the maximum depth of the execution sequences from the Event-B specification. "-" indicates that no restriction is applied to the range.

As shown in Table II, checking the different behaviors of the OS requires to use different bounds; therefore, when we extend ranges for checking specific properties, we need to perform the boundary check.

TABLE II: Estimated Bounds

Prp.	V: `tasks`, `tpri`, `res`, `rpri`, `evt`, `inr`, `ipri`	E	D
P1	`{T1,T2}`, `{1,2}`, {}, {}, {}, {},{}	AT,TT	-
P2	`{T1,T2}`, `{1}`, {}, {}, {}, {},{}	DT,AT,CT,TT	-
P3	`{T1,T2}`, `{1}`, {},{}, `{Evt1}`, {}, {}	DT,AT,WE,SE	-
P4,5	`{T1,T2,T3}`, `{1,2,3}`, `{Res1}`,`{6}`, {}, `{Inr1,Inr2}`, `{4,7}`	DT,DR,DI,AT, GR,RR,SI,RI	-
P6	`{T1,T2,T3,T4}`, `{10}`, {}, {}, {}, {},{}	DT,AT	-
P7	`{T1,T2}`, `{1}`, `{Res1}`, `{2}`,{}, {},{}	DT,DR,DI,AT, GR,RR,CT,TT	-
P8	`{T1}`, `{1}`, {}, {}, {},{},{}	DT,AT	-

Verification Results. All experiments were conducted on an Intel(R) Core(TM) i7 Processor at 2.67GHz running Linux. We performed experiments in two phases. In phase (I), we checked the OS design with respect to various properties. No bug was detected in the design. This result was considered due to the fact that the OS design had already been reviewed carefully by many researchers and engineers. Still, from this result, we can be sure that the OS design is correct with respect to its specification within input bounds. However, such successful experiment results do not show the effectiveness of the approach. We evaluate the effectiveness of the approach based on its bug-detecting ability. To show the bug-detecting ability of the approach, in phase (II), we intentionally added several bugs into the OS design and performed experiments to check the modified OS design with respect to the same properties as those verified in phase (I). Our purpose here is to make sure our approach can actually detect the bugs we added. In this paper, we focus on two kinds of typical bugs of the OS design: (i) the bugs that cause the condition enabling the OS services not to conform to the specification; and (ii) the bugs that cause the computational effects provided by the OS services to violate the specification. In practical environments, such kinds of bugs could be easily added into the design.

We present the results of experiments in Table III, which are outputted by Spin. Here, the first column ("No.") represents experiment number. Cases I.1-I.8 belong to phase (I) and cases II.1-II.8 belong to phase (II). Column "Prp." refers to the properties of the OS. Column "Bounded Ranges" represents bounds used in distinct experiments, where "t", "tp", "r", "rp", "i", "ip", and "e" present the size of ranges for instances of tasks, resources, events, interrupt routines, and ranges for values of the priorities assigned to tasks, resources, and interrupt routines, respectively. In column "*E*", we show the size of the restricted set of system services that are required for checking the corresponding properties. Column "*D*" presents the maximum depth of execution sequences from the Event-B specification. "-" indicates that no restriction is applied to the range. Column "LTS Generation" shows statistics of the LTS generator. Here, columns "#State", and "#Trans" present the number of distinct states and that of transitions appearing in the LTS; column "Time" presents the time taken (s) for the generation. Column "Model Checking" presents statistics of the model checker including actual memory usage for distinct verification, the time taken (s), and the verification result in which "$\surd$" indicates the successful result - the verification has completed and no bug has been found in the design, and "Fail" indicates that some bugs have been found in the design.

Firstly, we use the least ranges which reflect the appropriate configuration to check the desirable behaviors for the enumerated properties. The verification outputs for such cases are presented in rows I.1, I.2-1, I.3, I.4, I.5-1, I.6, I.7, and I.8. Then, we extend ranges gradually so that the verification covers many more behaviors than those focused in previous steps as long as the machine capacity allows this. For example, the ranges used in cases I.2-2 and I.2-3 are extensions of those in I.2-1; and, the ranges used in case I.5-2 are extensions of those in I.5-1.

In the cases of I.5-3, I.5-4, and I.5-5, we want to check the interaction of multiple system services; therefore, we do not restrict the system services in these experiments, i.e. E is presented as "-" in the table. However, within the given V, the LTS of the specification is huge. We restrict the depth of the execution sequences of the specification to reduce the size of LTS. We give values for the maximum depth based on the estimation of the size of LTS that the machine capacity allows. For example, with currently used machines, we estimated that the machine capacity allows round 20000 transitions appearing in LTS. Firstly, we generate LTS with the given ranges for V and no limitation for the depth of the execution. Then, if LTS is huge, we try some values for the depth. For example, if we use 7 tasks, 1 resource, 2 interrupt routines, 1 event, and depth = 7, then "#Trans" = 14046, the verification succeeds because the size of LTS is less than that the machine capacity allows. However, when we use the same ranges for V and depth = 8, "#Trans" = 32599, the size of LTS is significantly large compared with the used machine capacity. This could easily cause the state explosion. Therefore, we set the depth to 7 in our experiments as shown in the case of I.5-4.

From the experiment results, we can see that the time taken and the total actual memory usage for the LTS generation and the model checking are reasonable. For the model checking result of phase (I) shown in the table, no bugs were reported in all cases of experiments. However, in phase (II), some bugs were intentionally added into the OS design. Consequently, they were shown in the model checking results of the modified OS design. From the experiment results, we can see that bugs were detected in short time and with reasonable total actual memory usage for the model checking.

We added a bug to the condition expression for waking up the task waiting for an event in function `SetEvent`. This bug is detected in case II.3. A counter-example is shown by Spin against (P3): `ActivateTask(t1); ActivateTask(t1,t2); WaitEvent(t2, evt1); GetResource(t1,r1); ActivateTask(t1,t3); SetEvent(t3, t2, evt1)`: text of failed assertion: $assert((tsk\_state[1].tstat == 2))$. Task `t2` in the waiting state is not released to the ready state, even though `evt1` has been set for `t2` by tasks `t3`. This violates the specification. Because the condition expression for waking up the task waiting for an event in the OS design includes bugs, `t2` does not satisfy this condition. Thus, `t2` is not waked up.

We added another bug in the computational statement of function `GetResource`. This bug makes the dynamic priority of tasks not change for any case when the tasks get the resources. This bug is detected in cases II.4, II.5, and II.7. In case II.4, a counter-example is shown by Spin against (P4): `ActivateTask(t1); ActivateTask(t1,t2); WaitEvent(t2,evt1); GetResource (t1,r1)`: text of failed assertion: $assert(tsk\_state[0].dpriority == 6)$. The priority of task `t1` is not raised to the ceiling priority of resource `r1`, even though `t1` has got `r1` successfully. This is because the ceiling priority of resource `r1` is not assigned to the dynamic priority of task `t1` even though the static priority of `t1` is lower than the ceiling priority. This violates the specification.

Experiment results above show the ability of our approach to detect the typical bugs of the OS design such as the bugs in the guard conditions enabling the computations and those in computational statements of functions. Such bugs cause undesirable design behaviors of the OS. With the exhaustive verification within the bounds, our approach provides rapid bug detection of design behavior.

VI. Discussion

Bounds. In the framework, the bounds were defined to restrict the range of every data element including variables, constants and parameters. We can obtain a finite LTS associated to the Event-B specification within the bounds. Basically, the restriction of every data element produces a finite representation of the target system; this makes possible to apply the model checking technique. However, the OSEK/VDX operating systems includes several data elements and several functionalities, the size of the LTS may be so large that it could easily cause the state explosion when we apply model checking, event though the LTS is finite. To avoid the state explosion, our idea is to lead the verification to focus on partial behaviors of target systems. We additionally define restriction of service of the target system and the depth of the execution as well. With these restrictions, we can check each functionality of

TABLE III: Experiment Outputs

No.	Prp.	Bounded Ranges						LTS Generation			Model Checking		
		V				E	D	#State.	#Trans	Time(s)	Mem(Mb)	Time(s)	Result
		t,tp	r,rp	i,ip	e								
I.1	P1	2,2	0,0	0,0	0	2	-	4	10	1.0	129.2	3.5	√
I.2-1	P2	2,1	0,0	0,0	0	3	-	4	10	1.0	129.2	3.5	√
I.2-2	P2	9,3	0,0	0,0	0	4	-	512	13824	3.5	430.0	362.3	√
I.2-3	P2	5,5	0,0	2,2	0	6	-	128	1536	2.2	133.5	17.6	√
I.3	P3	2,1	0,0	0,0	1	3	-	10	27	1.0	129.2	4.2	√
I.4	P4	3,3	1,1	2,2	0	6	-	80	520	1.3	129.2	8.3	√
I.5-1	P5	3,3	1,1	2,2	0	6	-	80	520	1.3	129.2	8.3	√
I.5-2	P5	3,3	1,1	2,2	1	12	-	152	1036	2.0	132.5	14.8	√
I.5-3	P5	7,3	1,1	2,2	1	-	6	374	7063	2.5	227.6	105.7	√
I.5-4	P5	7,3	1,1	2,2	1	-	7	731	14046	3.6	285.4	360.0	√
I.5-5	P5	10,3	1,1	2,2	1	-	4	335	9476	10.0	413.8	75.4	√
I.6	P6	4,10	0,0	0,0	0	2	-	102	1220	2.1	132.0	15.0	√
I.7	P7	2,1	1,1	0,0	0	5	-	8	22	1.0	130.0	7.2	√
I.8	P8	1,1	0,0	0,0	0	1	-	2	2	1.0	129.2	3.5	√
II.1	P1	2,2	0,0	0,0	0	2	-	4	10	1.0	129.2	3.5	√
II.2	P2	2,1	0,0	0,0	0	3	-	4	10	1.0	129.2	3.5	√
II.3	P3	2,1	0,0	0,0	1	3	-	10	27	1.0	129.2	4.2	Fail
II.4	P4	3,3	1,1	2,2	0	6	-	80	520	1.3	129.2	6.0	Fail
II.5	P5	3,3	1,1	2,2	0	6	-	80	520	1.3	129.2	6.0	Fail
II.6	P6	4,10	0,0	0,0	0	2	-	102	1220	2.1	132.0	15.0	√
II.7	P7	2,1	1,1	0,0	0	5	-	8	22	1.0	129.2	5.2	Fail
II.8	P8	1,1	0,0	0,0	0	1	-	1	1	1.0	129.2	3.5	√

OSEK/VDX OS independently from the other functionalities, e.g., the task management is checked independently from the resource management, the event management, and the interrupt handling. We also can check each small groups of functionalities instead of all at one, e.g., checking the combination of task management, resource management and event management. Each of these groups represents some essential behaviors of the target system. We could distribute partial behaviors in variations of the environment. Accordingly, some questions may arise regarding how we decide the partial behaviors included in the verification and avoid the state explosion. In our idea, the partial behaviors are decided according to the properties of the target systems to be checked. Consequently, we found that one could avoid the state explosion if we use reasonable ranges for data elements and services of the OS. Even though we cannot show the conformance of the design and the specification in the infinite scope but if an error is returned within the bounds, we can show it really exists in the design.

Coverage. The coverage of this verification is evaluated by how much of the specification is satisfied by the design. In our experiments, the design is checked against the LTS which are generated from the specification within input bounds. There are two viewpoints to evaluate how much of specification is represented in the LTS. They are structure and behavior. Structure refers to a set of entities concerned with the configuration. Behavior refers to system services. Therefore, we divided the coverage criteria into two types: structure coverage means how large of the configuration is used in the verification; and behavior coverage refers to not only the individual function call but also the order of function calls. For structure coverage, we determine the bounds of the execution sequences based on the properties of interest. Specifically, we define the bounds at least as large as to cover the configuration appearing in the scenario corresponding to the property. For these bounds, we were able to check important properties of the OS within a reasonable time and memory space. To get more reliability in the verification, we need to extend the bounds as large as possible depending on the machine capacity. For behavior coverage, we have checked each functionality such as task management, resource management, event handling independently of others, including both of regular sequences and irregular sequences. Even though we cannot check all the functionalities at once due to the state explosion we still need to check at least the combination of functionalities such as the combination of resource management and event handling, and combination of event handling and interrupt processing. Checking this combination is important because it is known that bugs often come from the interaction of different functionalities. In order to check multiple functionalities at once while avoiding the state explosion, we need to make the bounds of configuration as small as possible. We consider that it is important to have a good balance between the ranges of structures and behaviors based on the properties to be checked.

VII. Related Works

[4] and [7] present case studies on checking the operating systems compliant with OSEK/VDX. In [4], the authors describe the properties of interest in temporal logic formulas and describe the design in Promela. In [7], the authors express the properties in terms of the first-order logic and model the OS as CSP process. In these works, the limited configurations are used in the experiments to apply the model checking; however, how to estimate appropriate configurations for the verification is not explained. By using Event-B, we easily specify the properties and the external behaviors of the OS and ensure the quality of the specification before using it to check the design. In addition, we present a way to determine the appropriate bounds for the verification of desirable properties.

[17] verifies the OS design by constructing a general model of the environment from scratch: it includes a class diagram and state diagrams of objects in the environment. These diagrams are composed to generate the environment scripts. In our work, the environment is generated from the Event-B specification. Hence, by construction, it is comprehensive with

respect to the specification. The environment is used to exercise the design and check the given relation between variables of the Promela design and variables of the Event-B specification in every reachable state. This shows that the design satisfies the specification. In our case study, the correctness of the specification is guaranteed by tools of Event-B; the quality of the environment is improved. Also, various ranges are customized to direct the verification focus on the behaviors relevant to intended properties and bugs.

[8] present an approach to verify the OS kernels based on theorem proving. Theorem proving can be used to verify the infinite systems; but, it generally requires a lot of interactive proofs. In our workflow, we use model checking combining with prover tools of Event-B. Although ranges are bounded due to the limitation of model checking; however, we are able to improve quality of the properties checked and get completely automatic verification. Therefore, we have a high degree of confidence in the verification results.

For combination of Event-B and model checking, tools such as ProB[9] and Eboc[10] work as model checkers for Event-B; [11] translates Event-B model into Promela model and uses Spin to check the model. These existing works focused on verifying the Event-B model. Separately, we use the Event-B specification to verify the design in Promela. In addition, we did not directly translate Event-B code into Promela but translated the LTS of the Event-B specification and assertions into Promela. Then, we input the combination between the OS design and the LTS of the specification into Spin to check a simulation relation between them.

VIII. Conclusion

We presented a case study of applying the existing framework to the verification of the OS design. In this application, we formalize the OSEK/VDX OS specification in Event-B and use it to verify the design of the OS. With rich notions, Event-B facilitates describing the specification of the OS. Specifically, it is convenient to specify abstract data structures and express the conditions for the correctness of system services in terms of pre-conditions and post-conditions by using such data structures. Moreover, we are able to ensure the consistency of the specification before inputting it in the model checker. Promela is intended to analyze the design of the OS using the Spin model checker. It is an appropriate model language to describe low-level data structures and the collaboration of internal components with highly optimized behaviors. The framework is straightforwardly applied to check the design of the OS in Promela with respect to the specification in Event-B. Three ranges can be effectively applied in Event-B. In addition, it is feasible to generate the LTS from the Event-B specification. This is a source to generate exhaustive sequences of function calls for verification of the design. The behaviors of the OS is deterministic; such a way of verifying the simulation relation between the specification and the design is sufficient to show that the OS design satisfies its specification. The results of the experiments demonstrate that this approach can be practically applied in verification of important properties and detection of typical bugs of the target system. This exhibits an ability to deal with the specifications and the designs which are described in different specification languages. Therefore, we can choose appropriate specification languages to describe the specification and the design for the purpose of verifying the design. As a result, one could decrease the cost for describing the specification and the design. Also, one could decrease the cost for the verification since the verification is executed automatically with the given bounds and mappings. In the future, we aim at an effective mechanism to identify and remove the symmetric variations of the initial states of the systems checked.

Acknowledgments

This work is supported by the project No. 102.03–2015.25 granted by Vietnam National Foundation for Science and Technology Development (Nafosted).

References

[1] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge Univ Press, 2010.

[2] T. Aoki, “Model checking multi-task software on real-time operating systems,” in *The 11th IEEE International Symposium on OO Real-Time Distributed Computing*, 2008, pp. 551–555.

[3] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[4] Y. Choi, “Model checking trampoline os: a case study on safety analysis for automotive software,” *Softw. Test., Verif. Reliab.*, vol. 24, no. 1, pp. 38–60, 2014.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification,” in *Proceedings of the 21st International Conference on Softw. Eng.*, 1999, pp. 411–420.

[6] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[7] 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,” in *Theoretical Aspects of Software Engineering*, Aug 2011, pp. 142–149.

[8] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: Formal verification of an operating-system kernel,” *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, 2010.

[9] M. Leuschel and M. Butler, “ProB: An automated analysis toolset for the B method,” *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 2, pp. 185–203, 2008.

[10] P. Matos, B. Fischer, and J. Marques-Silva, “A lazy unbounded model checker for event-b,” in *Formal Methods and Softw. Eng.*, 485-503, 2009, vol. 5885.

[11] T. MULLER, “Formal methods, Model-Cheking and Rodin plugin development to link Event-B and SPIN,” 2009.

[12] OSEK/VDX Group, “OSEK/VDX operating system specification 2.2.3, http://portal.osek-vdx.org/.” [Online]. Available: http://portal.osek-vdx.org/

[13] RODIN and DEPLOY group, “Event-B and the RODIN platform, http://www.event-b.org/.” [Online]. Available: http://www.event-b.org/

[14] M. Y. Vardi and P. Wolper, “An automata-theoretic approach to automatic program verification,” in *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, 1986, pp. 332–344.

[15] D. H. Vu and T. Aoki, “Faithfully formalizing OSEK/VDX operating system specification,” in *Proceedings of the 3rd Symposium on Information and Communication Technology*, 2012, pp. 13–20.

[16] D. H. Vu, Y. Chiba, K. Yatake, and T. Aoki, “Model checking conformance of a promela design to its formal specification in Event-B,” in *The third International Workshop on Formal Techniques for Safety-Critical Systems*, 2014, pp. 203–218.

[17] K. Yatake and T. Aoki, “Model checking of OSEK/VDX OS design model based on environment modeling,” in *Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing (ICTAC ’12)*, 2012, pp. 183–197.