Constraint Specification and Test Generation for OSEK/VDX-Based Operating Systems*

Yunja Choi

School of Computer Science and Engineering, Kyungpook National University, Korea
yuchoi76@knu.ac.kr

Abstract. This work suggests a method for systematically constructing an environment model for automotive operating systems compliant with the OSEK/VDX international standard by introducing a constraint specification language, OSEK_CSL, and defining its underlying formal models. OSEK_CSL is designed for specifying constraints of OSEK/VDX using a pre-defined set of constraint types identified from the OSEK/VDX standard. Each constraint specified in OSEK_CSL is interpreted as a context-free language and is converted into push-down automata using NuSMV, which allows automated test sequence generation using LTL model checking. This approach supports selective applications of constraints and thus is able to control the "degree" of test sequences with respect to test purposes. An application of the suggested approach demonstrates its effectiveness in identifying safety problems.

1 Introduction

An automotive operating system is typical safety-critical software and therefore requires extensive analysis using formal methods. However, existing formal approaches in this domain [6,7] have either been seen difficult to use or do not scale in practice. Instead, conformance testing [12] has been a de facto verification method in industry; for example, in order to get a certificate for an operating system compliant with the OSEK/VDX international standard [1], a system must pass a test suite distributed by a certification agency. A major problem with conformance testing is that the tests are designed for checking functionalities, not for checking safety, and do not aim at comprehensive verification. Our previous work [4] revealed some potential safety problems in an OSEK/VDX-based operating system using model checking, which would have slipped through with conformance testing.

Though a comprehensive but cost-effective verification approach is hard to find, we may be able to control the degree of comprehensiveness by modularizing systems and selectively applying verification techniques so that we can achieve comprehensiveness to an anticipated degree with moderate cost. This work aims at automated test sequence generation that allows comprehensive

* This work was supported by the National Research Foundation of Korea Grant funded by the Korean Government (NRF-2012R1A1A4A01011788).

R.M. Hierons, M.G. Merayo, and M. Bravetti (Eds.): SEFM 2013, LNCS 8137, pp. 305–319, 2013.
© Springer-Verlag Berlin Heidelberg 2013

checking of possible interactions between an automotive operating system and its application tasks. This is achieved by (1) introducing a constraint specification language, *OSEK_CSL*, designed for specifying constraints identified from the OSEK/VDX standard, (2) defining its underlying formalism in pushdown automata whose formal models are modularly defined in the input language of the symbolic model checker NuSMV [13], and (3) generating test sequences using LTL model checking.

OSEK_CSL is devised to make the specification of operational environments modular and systematic; it is a simple and intuitive constraint specification language consisting of only four basic building blocks, each of which can be independently specified and imposed on a system model. Each constraint specified in OSEK_CSL is systematically translated into NuSMV and combined with a generic task model. The task model is pre-defined as a NuSMV module representing the abstract behavior of a generic task as required in the standard. It is an abstract task model since it includes only the basic requirements from the OSEK/VDX standard without any implementation details. We have standardized the mapping between each constraint and a NuSMV module so that any number of constraints can be added by engineers and their corresponding NuSMV modules can be instantiated automatically.

Test sequence generation is automated through LTL model checking on the NuSMV model using trap properties designed to cover all transitions for each constraint/task module. Our approach enables us to control the degree of test sequences from "perfect" to "erroneous" and "false", depending on the number of constraints imposed for LTL model checking. In this way, the generated test sequences include correct inputs as well as undesirable or unexpected inputs, as required by safety analysis.

Our approach is applied to Trampoline [2], an open source operating system based on OSEK/VDX, and identified two assertion violations and a segmentation fault error that had been missed by existing approaches, including conformation testing and model checking.

The remainder of this paper is organized as follows: Section 2 briefly sketches the background of this work and the overall approach. Section 3 introduces our OSEK_CSL language. Section 4 explains the NuSMV module for a representative OSEK_CSL constraint type and the test sequence generation approach using LTL model checking. An application result using the suggested approach is presented in Section 5, followed by a discussion on related work (Section 6) and the conclusion (Section 7).

2 Background and Approach

OSEK/VDX is a joint project of the automotive industry, which aims at establishing an industry standard for an open-ended architecture for distributed control units in vehicles. The standard has been adopted by major automobile manufacturers as well as by the AUTOSAR open source architecture defined by a consortium of over 50 automotive manufacturers worldwide.

Conformance testing is a standard verification method for the certification of OSEK/VDX-based operating systems. However, conformance test suites are typically insufficient for identifying safety problems. As OSEK/VDX explicitly specifies more than 26 basic APIs, thorough conformance testing would require at least $26 \times 2 \times 3$ test cases, even if we assume two arguments per API and even if only boundary values for the arguments were chosen. The possible number of execution sequences for these $26 \times 2 \times 3$ test cases would rise to 156 factorials, a large number to be tested in practice.

Our previous works tried to address this issue using property-based code slicing and test generation [4,14]. The idea was to perform focused verification by slicing the operating system kernel with respect to the given safety properties. Those approaches have proven increased verification efficiency and effectiveness in identifying safety issues. Nevertheless, model checking still costs a lot (e.g., 30 Gbytes of memory were consumed during verification of one safety property) and requires some knowledge of the underlying technique. Property-based slicing and test generation were cheaper and easier to apply in practice compared to a similar approach using model checking, but comprehensiveness was not achieved. Both cases over-approximated the system environment by allowing non-deterministic API calls from tasks and by informally imposing constraints on the environment model or during the scenario generation process.

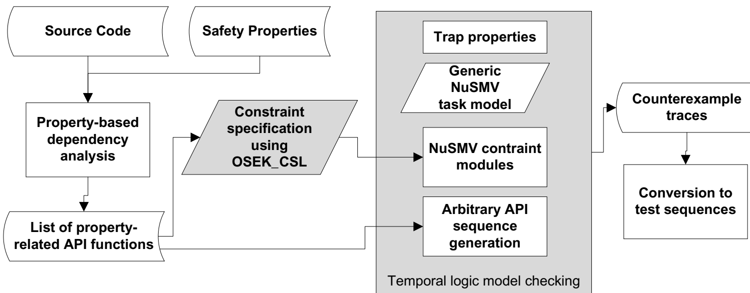

Fig. 1. Overall approach

Figure 1 illustrates an overview of our improved approach. First, we optionally identify API functions related to given safety properties through a dependency analysis using a static code analyzer, as explained in [14]. This process is not mandatory, but helpful in reducing the test input space. Once the list of (property-related) API functions has been determined, users need to specify the desired system constraints using OSEK_CSL and choose which of those constraints will be actually imposed during test sequence generation. OSEK_CSL consists of four basic constructs, each representing a constraint type. We have defined an NuSMV module for each construct as a pushdown automaton. A generic task model for OSEK/VDX-compliant operating systems is also predefined as an NuSMV module. Finally, a set of trap properties is specified in LTL by asserting that not every state or transition is reachable in each NuSMV module.

Temporal logic model checking is performed to verify whether the given trap properties are satisfied by the model, defined as a conjunction of the generic task model, the set of constraint modules, and arbitrary API sequences. If a trap property is refuted, the corresponding counterexample sequence is converted into a test sequence.

Since the first part of the approach was already explained in [14] and the approach is independent of whether property-based extraction of API functions is used or not (entire API functions can be used as they are), this paper focuses on constraint specification using OSEK_CSL, constraint modeling in NuSMV, and test sequence generation using temporal logic model checking.

3 Constraint Specification Language

A typical and straightforward environment of an operating system is an arbitrary call sequence of API functions provided by the operating system, which apparently simulates an actual environment, but includes too many impossible or undesirable interactions and results in a large number of false alarms when verification is performed. Analyzing counterexamples and identifying false alarms is a time-consuming process. To reduce such inefficiency, this work suggests a systematic method for formalizing constraints from the OSEK/VDX standard and reflecting them in the environment model.

3.1 OSEK/VDX Requirements and Constraints

OSEK/VDX defines task models for user-defined tasks, which are the basic building blocks of an application program. A task interacts with the operating system through system calls. OSEK/VDX explicitly defines a total of 26 such APIs. Figure 2 (a) is the task model for an extended task specified in the standard. Figure 2 (b) is our version of the same task model annotated with related APIs and an explicitly specified initial state. We use three types of annotations; the one finishing with '?' represents an external API call from other tasks, the one finishing with '!' is an internal API call, and the one surrounded by <> is an internal event caused by system scheduling. For example, the transition from *running* to *suspended* is triggered by the internal API call to either $TerminateTask$ or $ChainTask$, but the transitions between *ready* and *running* are caused by priority-based task scheduling.

The OSEK/VDX standard explicitly/implicitly specifies constraints among the APIs, some of which are listed in Table 1. Analyzing those constraints reveals that they can be categorized into four types.

1. A system call f_1 shall be followed by f_2 (though not necessarily directly).
2. The number of calls to f is limited by n.
3. A system call f shall not be called in between two system calls f_1 and f_2.
4. No system call shall be made after a call to f.

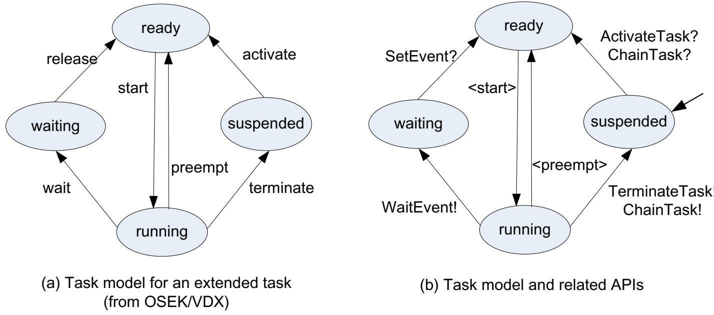

Fig. 2. Task model and related APIs

For example, if *GetResource* is called, the matching system call *ReleaseResource* must be called afterwards, and *WaitEvent* shall not be called in between them.

The types of constraints can also be classified by scope since some constrain global behavior and others constrain local behavior; *GetResource* and *ReleaseResource* are in the local scope because once a task calls the *GetResource* system call, *ReleaseResource* needs to be called in the same task. On the other hand, *WaitEvent* and *SetEvent* are in the global scope. The task that calls *SetEvent* should be different from the task that calls *WaitEvent* for the same event, but they need to be called in pairs. *TerminateTask* and *ChainTask* are the examples that cannot be followed by any system calls in the same task.

3.2 Constraint Specification Language OSEK_CSL

To formally specify such constraints, we define a simple constraint specification language called OSEK_CSL (Constraint Specification Language for OSEK/VDX). OSEK_CSL consists of four basic constraint types, which can be defined with context-free grammars and their corresponding pushdown automata. This section introduces each basic constraint type, defines the constraint

Table 1. Constraints from the OSEK/VDX standard

	Constraints
C1	Ending a task without a call to TerminateTask or ChainTask is strictly forbidden and causes undefined behavior.
C2	TerminateTask, ChainTask, Schedule, WaitEvent shall not be called while a resource is occupied.
C3	A task calling WaitEvent shall go to the waiting state and shall not be activated. again before SetEvent is called by other tasks.
C4	OSEK strictly forbids nested access to the same resource.
C5	A task shall not terminate without releasing resources.

specification language, and provides formal definitions for representative constraint types.

Definition 1 *(constraint types) Let Σ be a set of API functions in the OSEK/VDX standard and N a set of natural numbers. For any $f, f_1, f_2 \in \Sigma$, $A' \subseteq \Sigma$, and $n \in N$,*

1. $InPairs(f_1, f_2)$ *:* f_2 *shall be called after for each call to* $f1$*.*
2. $Limited(f, n)$ *: The number of calls to* f *shall not exceed* n*.*
   - $SetLimited(A', n)$ *: The total number of calls to the functions in* A' *shall not exceed* n*.*
3. $NotInBetween(f, f_1, f_2)$ *: A call to* f *shall not be allowed in between calls to* f_1 *and* f_2*.*
4. $MustEndWith(f)$ *:* f *shall be called eventually and no calls shall be allowed afterwards.*

Each constraint type can be defined as a context-free language or a regular language over Σ. For example, $Limited(f, n)$ and $SetLimited(A', n)$ are regular languages that can be formalized using finite automata. $InPairs$, on the other hand, requires a little more thought since it cannot be expressed in regular language, as we need to keep track of the number of calls to a specific system call. In fact, the derivation rule for $InPairs(a, b)$ can be defined as follows:

$$S \to aSb \mid abS \mid Sab \mid xS \mid \lambda,\ x \notin \{a, b\}.$$

For $NotInBetween(c, a, b)$, where $InPairs(a, b)$ is true, the derivation rules $S \to aSb$ and $S \to xS$ are refined:

$$S \to aS'b \mid abS \mid Sab \mid xS \mid \lambda,\ x \notin \{a, b\}$$
$$S' \to yS' \mid S,\ y \notin C(a, b) \cup \{a, b\},$$

where $C(a, b) = \bigcup\{c \mid NotInBetween(c, a, b)\}$.

Internal formal specification of these constraints can be standardized as shown in Figure 3. Figure 3 (a) is a pushdown automaton for $InPairs(a, b) \wedge NotInBetween(c, a, b)$; s_0 is the initial state and the final state. It ignores letters other than a, moves to state s_1, pushing 0 to the stack once it receives a. In s_1, it ignores letters other than a, b, and c. It moves to s_2, pushing 1 into the stack, if it receives a. It pushes 1 for each input a, pops for each input b, does not change for each input other than a, b, c, and moves to s_0 if the input is b and the stack top is 0. Receiving input c when it is in state s_1 or s_2 results in moving to s_3, which is an error state.

Figure 3 (b) shows the formal representation of $InPairs(a, b) \wedge NotInBetween(c, a, b) \wedge Limited(a, n)$, limiting the size of the stack and checking whether the stack is full or not during the language process.

$SetLimited(A', n)$ allows us to specify the limit of the calls to a set of APIs. For example, $SetLimited(\{f_1, f_2\}, 10)$ specifies that the number of calls to f_1

plus the number of calls to f_2 shall not exceed 10, which can be categorized as a regular language. The rule for $MustEndWith(f)$ is also simple:

$$S \rightarrow xS \mid f, \text{where } x \in \Sigma - \{f\}$$

These constraints are classified into global constraint types and local constraint types. A global constraint type must hold in a global scope, i.e., among tasks, and a local constraint type must hold within a task. According to the OSEK/VDX standard, $Limited$ and $SetLimited$ are global, while $NotInBetween$ and $MustEndWith$ are local. $InPairs$ can be both. For example, $InPairs(WaitEvent, SetEvent)$ has global scope, but $InPairs(GetResource, ReleaseResource)$ has local scope. To distinguish global $InPairs$ from local ones, we add $GInPairs$ to the four basic constraint types in $OSEK_CSL$.

An environment of an OSEK/VDX-based operating system is defined using $OSEK_CSL$ based on these four constraint types.

Definition 2 *(Environment Model) The language induced by $OSEK_CSL$ is the intersection of an arbitrary number of languages defined by the basic constraint types. Formally, let L_i be a language defined by one of the constraint types, and suppose there are n such languages. Then,*

$$L(OSEK_CSL) = \bigcap_{i \in \{1..n\}} L_i.$$

This defines an environment of an OSEK/VDX-based operating system.

4 Formal Specification Using NuSMV

Since there can be a number of constraints, we need to compute their intersections in order to identify a language accepted by all specified constraints.

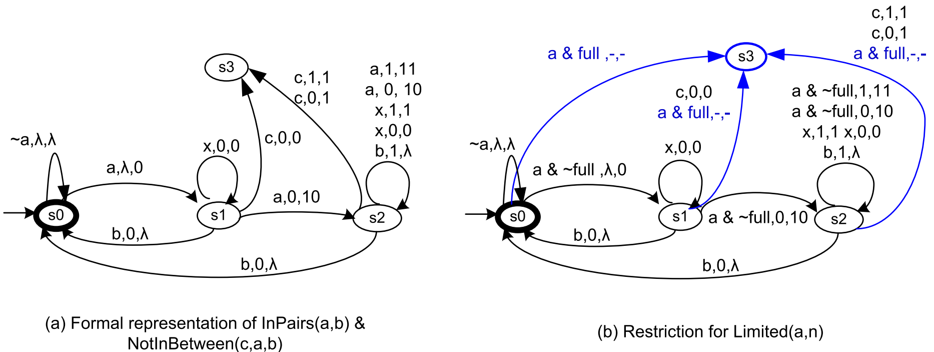

Fig. 3. Formal representation of constraint types

```
1 MODULE InPairs(first, second, alphabet, exclusiveSet, task)
2  VAR
3   state : {s0, s1, s2, s3};
4   mystack : STACK(first, second, alphabet, task.state, state);
5
6  ASSIGN
7   init(state):= s0;        /* initial state */
8   init(mystack.top):=0;    /* initial value of the stack top */
9    ...
10  next(state):=            /* defines the transition relation */
11    case task.state = running & !terminationRequested :
12      case state = s0  :
13             case next(alphabet)=first & mystack.top= 0      : s1;
14                  TRUE                                       : state;
15             esac;
16           state = s1 :
17             case  next(alphabet)=first & mystack[mystack.top]=0  : s2;
18                   next(alphabet)=second & mystack[mystack.top]=0 : s0;
19                   next(alphabet) in exclusiveSet                 : s3;
20                   TRUE                                           : state;
21             esac;
22        ...
23      esac;
```

Fig. 4. NuSMV MODULE for InPairs constraint type

We perform the computation using the model checker NuSMV under the assumption that the maximum number of system calls is bounded. This assumption is necessary since NuSMV is based on a finite state machine which cannot handle stacks of indefinite size as in constraint automata. Despite the limitation, NuSMV was chosen because of its modular structure, its simple but sufficient expression for specifying state machines, and most importantly, its powerful model checking capability for test sequence generation.

This section describes our method for modeling the representative constraint type using NuSMV by systematically mapping it to a MODULE in NuSMV. An NuSMV module for the generic task model is also introduced.

4.1 Formal Specification for Constrained Environments

Due to space limitations, this section provides details of the modeling approach only for the most frequently used constraint type $InPairs \wedge NotInBetween$. Figure 4 shows a fraction of the NuSMV MODULE for the constraint type.

MODULE InPairs(..) is a reusable component for specifying the constraint type $InPairs \wedge NotInBetween$ in the local scope. It is a straightforward translation of the automaton in Figure 3 (a); the parameters $first$ and *second* are for API function names that are supposed to be in pairs, *alphabet* is the set of

API names, *task* is the name of the task that is the scope of the constraint, and *exclusiveSet* is the set of API calls that are not supposed to be called in between *first* and *second*. The *exclusiveSet* is identified from the *NotInBetween* constraints. If there are no *NotInBetween* constraints related to *first* and *second*, then the *exclusiveSet* is empty and $InPairs \wedge NotInBetween = InPairs$. As in Figure 3 (a), there are four states s_0, s_1, s_2, and s_3, where s_0 is the initial state. The transitions between states are defined in the *ASSIGN* construct (line 6) using the *next* keyword. All the transitions are only possible when the task is in the *running* state. The model checks this condition by $task.state = running$ (line 11). The transition rules defined in *case..esac* are a direct translation of the pushdown automata in Figure 3 (a).

Whenever a new constraint of the $InPairs \wedge NotInBetween$ type is required, the module is instantiated in the NuSMV main module. For example, if there are two tasks with the same type of local constraints, we declare two constraints as follows:

$$constraint_1 : InPairs(f_1, f_2, alphabet, exclusiveSet, task_1);$$
$$constraint_2 : InPairs(f_1', f_2', alphabet, exclusiveSet', task_2);$$

The NuSMV module for the *InPairs* constraint type in the global scope differs little from that in the local scope, since the only difference between them is whether the stack of the pushdown automata is maintained locally or globally. Therefore, the same local module can be reused for global constraints of the same type. The signature for the global constraint type *GInPairs* is defined as

$$MODULE\ GInPairs(first, second, alphabet, exclusiveSet),$$

where its body is the same as that of *InPairs* except that line 11 of Figure 4 is removed. The task information is not passed to the global module since it is independent of tasks.

4.2 Formal Specification for Generic User Tasks

A Basic NuSMV Model for a User Task. A user task is modeled according to the task model shown in Figure 2 (b). Since it is a general task model, the same formal specification is used independent of the constraint types. The following shows the basic form of the NuSMV MODULE for the task model.

```
1 MODULE Task(alphabet, id, priority)
2  VAR
3    state : {suspended, ready, waiting, running};
4
5  ASSIGN
6    next(state):=
7      case
8        state = suspended  & (next(alphabet) = ActivateTask
9                             | next(alphabet) = ChainTask )   : ready;
10       state = ready   /* & if scheduled next */             : running;
```

```
11       state = running & (next(alphabet) = TerminateTask |
12                          next(alphabet) = ChainTask     )   : suspended;
13       state = running & next(alphabet) = WaitEvent          : waiting;
14       state = waiting & next(alphabet) = SetEvent           : ready;
15       TRUE                                                  : state;
16     esac;
   ...
```

However, this basic form is not enough to model task behavior, and it needs to be elaborated more to address the OSEK/VDX requirements. The following lists some of the requirements that have a direct impact on the task model;

1. A task transits to the *running* state from the *ready* state only if it is scheduled by the operating system (line 10). Priority-based FIFO scheduling is required in OSEK/VDX.
2. Only one task shall run at a given time. Line 10 should be constrained more to ensure this.
3. Task priority can be dynamically changed based on the PCP (Priority Ceiling Protocol). Since task scheduling is priority-based, the change of task priority needs to be specified.
4. PCP requires resource management. Therefore, we cannot correctly model task behavior without specifying resource management.

In order to address these requirements, the basic task model is refined by adding more abstract components and references to those components to the task model. For example, the signature of the task model changes to

```
MODULE Task(alphabet, id, priority, res1, res2, readyP, SomeoneIsRunning,
            conState),
```

where $res1$ and $res2$ are names of resources, $readyP$ is the name of the variable that keeps track of the highest number among the priorities of the tasks in the *ready* state, $SomeoneIsRunning$ is a global flag indicating whether there is a running task, and $conState$ is the state of the local constraint of the task. In order words, each task has references to resources, information on whether its constraints are currently satisfied or not, and some basic information about other tasks.

Handling Priority-based FIFO Scheduling. The generic task model keeps track of the state of each task and selects the task with the highest priority among all tasks in the ready state, instead of explicitly modeling a priority-based FIFO queue. This requires a simple change in line 10 of the basic task model:

```
10: state = ready & !SomeoneIsRunning :
       case priority >= readyP        : running;
            TRUE                      : state;
       esac;
```

It checks whether the state is ready and there is no task running currently. If this is true, it checks again whether the priority of the task is greater than or equal to all the priorities of the tasks in the ready state. The task moves to the running state only when all those conditions are satisfied.

Handling Priority Ceiling Protocol. The priority of a task is statically predefined for each task and cannot be changed throughout the whole execution life cycle, except for the case when it allocates a resource with higher priority than the task. This is called Priority Ceiling Protocol (PCP) and is designed to prevent the problem of priority inversion. The PCP is incorporated into the task model by defining transition rules for changing the priority of a task, depending on whether it allocates or releases resources. The following reflects the change of the basic task module when the system includes two resources:

```
next(priority):=
case state = running & next(alphabet)=GetResource :
   case (res1.owner=0 & (res1.priority > priority)) : res1.priority;
        (res2.owner=0 & (res2.priority > priority)) : res2.priority;
        TRUE                                       : priority;
   esac;
...
```

This model does not explicitly specify which resource is requested by which task, but models it as allocating whichever resource is available. It is originally required to specify the resource type when asking for resource allocation in the form 'GetResource(res1)', but our alphabet consists of API names without parameters for the sake of simplicity. For the input alphabet *GetResource*, it checks and allocates the first available resource. For *ReleaseResource*, it releases the last allocated resource first, as specified in the OSEK/VDX requirements.

4.3 Test Generation via LTL Model Checking

Given the generic task model and a set of constraints on the sequence of input alphabets, our goal is to generate task sequences w.r.t the API call sequence that executes all paths leading to either final states or error states of tasks and constraints. We define three types of trap properties for checking reachability:

Definition 3 *(trap properties) Suppose there are* n *number of constraints specified in OSEK_CSL and* m *tasks. Let* $CS^i = \{cs_0^i, cs_1^i, cs_2^i, cs_3^i\}, i = 1..n$*, be a set of states in the* i^{th} *constraint and* $S^j = \{suspended^j, ready^j, running^j, waiting^j\}, j = 1..m$*, a set of states in the* j^{th} *task. Then,*

1. *A trap property for checking whether there is a path from a* k^{th} *state to a final state in the* i^{th} *constraint/task:*

$$tp_{ik}^{cr} \stackrel{\text{def}}{=} G(CS^i.state = cs_k^i \rightarrow !\ F(CS^i.state = cs_0^i)) \text{ /*for constraints*/}$$

$$tp_{ik}^{tr} \stackrel{\text{def}}{=} G(S^i.state = s_k^i \rightarrow !\ F(S^i.state = suspended^i)) \text{ /*for tasks*/},$$

where $cs_k^i \in CS^i$, $s_k^i \in \{ready^i, running^i, waiting^i\}$.

2. *A trap property for checking whether an ith task can be activated at least twice:*

$$tp_i^{ta} \stackrel{\text{def}}{=} G((task_i.state \neq suspended^i \ \& \ X(task_i.state = suspended^i)) \rightarrow \\ !\ F(task_i.state = ready^i))$$

3. *A set of trap properties for checking whether each element of the alphabet is exercised as an input at least once:*

$$tp_\Sigma \stackrel{\text{def}}{=} \{G\ !\ (SomeoneIsRunning \ \& \ alphabet = a) \mid a \in \Sigma\}$$

Trap properties are specified in LTL (Linear Time temporal Logic), where the temporal connectives G, X and F mean "Globally", "neXt states" and "sometime in Future state", respectively. For example, tp_{ij}^{cr} means that "for all execution paths if the ith constraint is in state cs_j^i, there is no path from the state leading to the final state cs_0^i. tp_i^{ta} means that it is globally true that if the ith task is not in the *suspended* state and will transit to the *suspended* state in the next state, then there will be no path where the task reaches the *ready* state in the future. In other words, the property says that a task is not activated again once it is terminated. The set of system trap properties is the union of all three types of trap properties:

$$tp = \{tp_{ij}^{cr}, tp_{pq}^{tr}, tp_p^{ta} \mid i = 1..n, j = 0..3, p = 1..m, q = 1..3\} \cup tp_\Sigma.$$

Though we generate trap properties for all system constraints, the degree of constraints to be imposed on the model can vary. We define the *Degree of constraints (DoC)* as m/n, where n is the total number of constraints specified in OSEK_CSL and m is the number of actual constraints imposed on the NuSMV model. We say the environment model is perfect if $DoC = 1$, erroneous if $0 < DoC < 1$, and false if $DoC = 0$. We impose various degrees of constraints for counterexample generation because safety verification requires not only perfect test sequences, but also erroneous test sequences with illegal input values.

The set of trap properties is verified using the model checker NuSMV. NuSMV generates a counterexample trace if the properties are verified as false. The conversion from a counterexample trace to a test program is straightforward since the trace shows a step-by-step change of API calls for each task as shown in Table 2.

Table 2. A fraction of a counterexample trace for tp_{11}^{cr}

steps		1	2	3	4	5	6
$task_1$	state API calls	ready	running GetResource	running ReleaseResource	running GetResource	running WaitEvent	waiting
$task_2$	state API calls	running WaitEvent	waiting	waiting	waiting	waiting	waiting

Table 3. Comparison of branch coverage

	$schedule_r$	$schedule_d$	$schedule_w$	$schedule_s$	$event_w$	$task_s$	$event_s$
Formal	100%(3/3)	60%(3/5)	66.67%(2/3)	100%(1/1)	75%(3/4)	100%(4/4)	80%(4/5)
Informal	100%(3/3)	80%(4/5)	66.67%(2/3)	100%(1/1)	75%(3/4)	100%(4/4)	80%(4/5)

5 Experiments

A total of 30 counterexamples were generated for an OSEK/VDX model with two tasks, two local constraints, and one global constraint, using the suggested approach. It took 10 minutes 43 seconds for the whole counterexample generation, performing 44 iterations and searching $3.4e+10$ states for each LTL model checking process on average. The test sequences are used to test the OSEK/VDX-based open source operating system Trampoline [2], which was also used as a case example in our previous work using property-based code slicing and simulation-based scenario generation [14].

Table 3 shows the branch coverage of some of the Trampoline source functions identified by using property-based code slicing, comparing the coverage result using OSEK_CSL-based test sequence generation (Formal) and the result of using a random scenario generator (Informal)[1]. A total of 24 test sequences (after removing duplicated sequences) of an average length of 6 was used for the $Formal$ case, while one test sequence of length 32 was used for the $Informal$ case since it was the sequence that showed the best coverage from our previous work. Though we did not aim at high code coverage, Table 3 shows that the suggested approach achieves branch coverage similar to that of the best result using a random scenario generator.

The more interesting and important result is that the approach using $OSEK_CSL$ actually found safety problems that were missed throughout existing model checking and testing approaches. These include two assertion violations and one segmentation fault error. For example,

```
TASK(t1){                            TASK(t2){
  WaitEvent(e1);                          ReleaseResource(r1);
}                                         WaitEvent(e1);   }
```

is a test constructed from the counterexample trace generated from $G\ !\ (SomeoneIsRunning\ \&\ alphabet = ReleaseResource)$. This is an example of erroneous test sequences that do not obey constraints, since *ReleaseResource* is called without calling *GetResrouce* first. Running this test results in the following situation:

```
trampoline: ../os/tpl\_os\_kernel.c:522: tpl\_put\_preempted\_proc:
Assertion ‘tpl\_fifo\_rw[prio].size < tpl\_ready\_list[prio].size’
failed. ./doit: line 2: 25016 Aborted (core dumped) ./trampoline
```

[1] Abbreviated function names are used to save space.

These problems could not be identified by conformance testing or existing model-based test generation approaches because they are based on the "correct" model of OSEK/VDX and do not necessarily test illegal task behaviors.

6 Related Work

Specification-based test generation is a well-known technique. From the early 1990s, there have been numerous approaches that use formal languages to specify requirements and generate test cases [10]. Among them, references [3,9,16] are the closest to our work in that they also try to provide a solution for efficient verification of OSEK/VDX-based operating systems. [3] uses Z and SPIN for specifying test requirements and generating test sequences for OSEK/VDX. References [9] and [16] model OSEK/VDX requirements in Promela and generate test sequences by model checking trap properties using SPIN. All those works model OSEK/VDX functional requirements, but do not explicitly consider system constraints. Our work focuses on constraint specification in order to generate a more efficient interaction environment and provides modular specification methods for constraints.

Our work is also closely related to automated environment generation for software verification in general [5,15]; Tkachuk et al. [15] developed the Bandera Environment Generator, which automates the generation of environments from user-specified assumptions for Java programs. The specification is limited to regular expressions.

There have been more traditional approaches for verifying automotive software using formal methods [8,11], formally specifying OSEK/VDX requirements in CSP and performing formal verification using model checking or theorem proving. Using such formal specification languages requires experts in formal methods, who are usually not available in practice. Our approach provides an intuitive specification language with underlying formal specification so that constraints can be easily specified, transformed, and checked.

7 Conclusion

This work presents a systematic and modular method for specifying constraints for OSEK/VDX-based operating systems. Constraint specification plays an important role in constructing the correct environment of a system, enabling us to generate effective test sequences. We have analyzed types of constraints in OSEK/VDX, categorized them into four basic types, and defined an NuSMV module for each constraint type so that any constraint of a given type can be automatically instantiated. These constraints can be selectively imposed on the generic task model, generating varying degrees of test sequences. Comprehensiveness can be controlled through trap properties.

The suggested approach is extensible; though we have identified four constraint types in OSEK/VDX, there can be more. We believe that incorporating additional constraint types will not affect the existing definitions and the underlying formalism.

We note that NuSMV is an effective tool suitable for our purposes, but cannot handle infinite systems directly without abstractions. Future work will include more investigation on formal verification tools for infinite systems aimed at possible replacement of NuSMV and more extensive experiments with various measures.

References

1. OSEK/VDX operating system specification 2.2.3
2. Trampoline – opensource RTOS project, http://trampoline.rts-software.org
3. Chen, J., Aoki, T.: Conformance testing for OSEK/VDX operating system using model checking. In: 18th Asia-Pacific Software Engineering Conference (2011)
4. Choi, Y.: Safety analysis of the Trampoline OS using model checking: An experience report. In: Proceedings of 22nd IEEE International Symposium on Software Reliability Engineering (2011)
5. de la Riva, C., Tuya, J.: Automatic generation of assumptions for modular verification of software specifications. Journal of Systems and Software (2006)
6. In der Riden, T., Kanpp, S.: An approach to the pervasive formal specification and verification of an automotive system. In: Proceedings of the International Workshop on Formal Methods in Industrial Critical Systems (2005)
7. Lettnin, D., et al.: Semiformal verification of temporal properties in automotive hardware dependent software. In: Proceedings of Design, Automation, and Test in Europe Conference and Exhibition (April 2009)
8. Shi, J., et al.: ORIENTAIS: Formal verified OSEK/VDX real-time operating system. In: IEEE 17th International Conference on Engineering of Complex Computer Systems (2012)
9. Fang, L., et al.: Formal model-based test for AUTOSAR multicore RTOS. In: Proceeding of the IEEE 5th International Conference on Software Testing, Verification and Validation, pp. 251–259 (2012)
10. Hierons, R.M., et al.: Using formal specifications to support testing. ACM Computing Surveys (2009)
11. Zhao, Y., et al.: Modeling and verifying the code-level OSEK/VDX operating system with CSP. In: 5th International Symposium on Theoretical Aspects of Software Engineering, pp. 142–149 (2011)
12. John, D.: OSEK/VDX conformance testing - MODISTARC. In: Proceedings of OSEK/VDX Open Systems in Automotive Networks (1998)
13. NuSMV: A New Symbolic Model Checking, http://nusmv.irst.itc.it/
14. Park, M., Byun, T., Choi, Y.: Property-based code slicing for efficient verification of osek/vdx operating systems. In: First International Workshop on Formal Techniques for Safety-Critical Systems (2012)
15. Tkachuk, O., Dwyer, M.B., Pasareanu, C.S.: Automated environment generation for software model checking. In: 18th IEEE International Conference on Automated Software Engineering, pp. 116–129 (October 2003)
16. Yatake, K., Aoki, T.: Automatic generation of model checking scripts based on environment modeling. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 58–75. Springer, Heidelberg (2010)