A configurable V&V framework using formal behavioral patterns for OSEK/VDX operating systems∗

Yunja Choi∗
School of Computer Science and Engineering, Kyungpook Natl. Univ., South Korea

A R T I C L E   I N F O
Article history:
Received 18 April 2016
Revised 5 July 2017
Accepted 27 July 2017
Available online 2 August 2017

Keywords:
V&V framework
Formal patterns
OSEK/VDX
Embedded software

A B S T R A C T
Verification and Validation (V&V) of small-scale embedded software must consider the operating system. Unlike general-purpose systems, the underlying operating system is closely coupled with the application logic, generating potentially an infinite number of different control programs depending on the application configuration and application logic. Verifying this software individually is time-consuming and costly, especially when the objective is rigorous verification.

To assist in rigorous V&V activities for such embedded software, the proposed work suggests a pattern-based framework that can be used to generate configurable formal OS and test models. At the core of the framework, lies a set of predefined behavioral patterns and constraint patterns that can be composed for the auto-generation of formal models for variously configured operating systems. These configurable formal models form the basis of formal validation and verification activities such as model checking safety properties, model-based test generation, and formal application simulation. We have implemented a prototype tool, specially designed for embedded control software based on the OSEK/VDX international standard, to demonstrate the benefits of the framework in task simulation, test generation, and formal verification. A series of experiments and analysis demonstrate that the suggested pattern-based framework is more efficient in test sequence generation and more effective in identifying problems compared to existing approaches.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

The development of small-scale embedded software, such as control software for wearable devices or automotive Electronic Control Units (ECUs), normally follows a standard process: given an underlying operating system, developers define system configuration, such as the number of tasks and the number of resources to be used, and design an application logic that will be implemented and compiled with the required part of the operating system depending on the configuration.

Typically, the same operating system is used for multiple ECUs once it is developed according to an international standard, such as OSEK/VDX or AUTOSAR; however, a different control program is developed for each ECU depending on its role. Therefore, software developers must ensure that the control program is compliant with its underlying operating system to guarantee the safe operation of the control software and the correctness of the control logic. Our previous work (Choi and Byun, 2017; Byun and Choi, 2015) demonstrated that a control program that is not compliant to its underlying operating system could result in system failure.

Nevertheless, a common Verification and Validation (V&V) practice in this domain limits the target of V&V to only application software, assuming that the underlying operating system is correct (Ray et al., 2009; Marinescu et al., 2009; Delange et al., 2010; Berry, 2007; O’Halloran, 2013); this bypasses the complexity caused by operating systems. Although this approach makes sense when the underlying operating system is fully verified, it still excludes potential issues that may be caused by incorrect interactions between the operating system and application code. We are susceptible to untrustworthy or meaningless V&V results due to either the locality of the result or a large number of false alarms if we under-approximate or over-approximate the underlying operating system.

A major difficulty for more rigorous verification lies in the growing complexity of embedded software, characterized by application configurations, application logic, and the operating system kernel. A model-based approach can be a solution to this problem, where the common behavior of the operating system is formally
modeled and fully verified (Botaschjan et al., 2008; Endres et al., 2010; Klein et al., 2010; Zhu et al., 2013; Choi, 2014). The verified model replaces the actual implementation of the operating system when the focus of the V&V is on the application logic or on the interactions among applications and operating systems. Though this approach reduces the complexity of V&V, formal models are known to be difficult to construct, requiring expertise in both the application domain and formal methods, which is rare in practice.

Further, different application configurations may require different formal models of the underlying operating system to reflect the fact that only the part of the operating system kernel required by the configuration is compiled into the embedded software.

To assist in rigorous yet cost-effective verification and validation, we propose a pattern-based V&V framework. The basis of the proposed framework is a formal model of the operating system that is reconfigurable and auto-constructible. A set of formal behavioral patterns for each basic construct of the operating system is defined as a parameterized state machine, which can be assembled according to the system configuration. This enables the auto-generation of a configuration-dependent formal model of the operating system that can be used to validate and verify the embedded software constructed for each combination of configuration, application, and operating system. This configurable formal models can be utilized in various manners. It forms a basis for a comprehensive test sequence generation together with constraint patterns we have defined in our previous works (Choi and Byun, 2017; Byun and Choi, 2015), works as a formal simulation engine to validate task design of an application program, and can also be integrated with application code for formal verification.

To demonstrate the benefit of using the proposed pattern-based V&V framework, we have implemented a prototype toolset targeted at configurable test sequence generation for operating systems compliant to OSEK/VDX international standard. We then use this toolset to demonstrate applications of the framework in model-based test generation through a series of experiments.

The remainder of this paper is organized as follows: Section 2 motivates the work and the overall approach. Section 4 introduces the formal definitions of parameterized patterns for OSEK/VDX operating systems after defining the underlying formalism used in this work in Section 3. Section 5 summarizes the formal constraint patterns for modeling operational environments, which was published in Choi and Byun (2017) and Byun and Choi (2015), before explaining the concept of pattern-based verification and validation using an implementation of the suggested framework in Section 6. Applications of the framework are demonstrated through a set of experiments on the test generation for operating system implementations in Section 7. We conclude this paper in Section 9 after a brief discussion of related work in Section 8.

2. Motivation and approach

The goal of software safety is the avoidance of system safety failures which are caused by software errors and/or are detected and handled by software procedures (Leveson and Harvey, 1983). To ensure software safety, we need to make sure that the software performs a required function and that no specification errors, and no design and coding errors exist in the software. Naturally, software safety is a pre-requisite for the safety of the system controlled by the software.

Fig. 1 presents a software perspective of an embedded system. The embedded software is generated by compiling three components together: (1) system configuration, (2) application software, and (3) operating system kernel. The development of safety-critical application software primarily follows the model-driven approach performing V&V activities on the model level using tools such as

Simulink and SCADE (O’Halloran, 2013; Berry, 2007); The model-driven approach constructs design models of application software, validates their functional behavior through model simulation, verifies safety properties¹ through model checking, and ensures functional correctness with code generation from the verified model. However, the operating system part of the software is developed manually without tool support or guidelines for the V&V activities. The same is true for the combination of application code and the operating system kernel. Though model-based V&V for application software can be effective for ensuring the safety of unit programs, it is not possible to verify behaviors that may be affected by interactions with the underlying operating system, such as task scheduling, interrupts, and alarms, without considering the underlying operating system. Extensive testing is performed on system level (e.g. HiL testing) focusing on functional correctness; However, it is difficult to evaluate the test adequacy with respect to the software. The following summarizes the major characteristics of embedded software.

1. It is configuration-dependent: A configuration set defines the final application binary, even though a common kernel code is used. V&V is required not only for the common OS kernel, but also for each combination of configuration, application, and kernel.

2. The safety of embedded software is affected by that of the OS kernel code, application code, and safe interactions between the application and kernel. In particular, the software interaction portion must be thoroughly verified before testing the system as a whole.

3. The application binary is primarily platform-dependent and the system platforms are typically limited in hardware resources.

4. Supporting tools for safety verification or validating embedded software are rare.

The proposed approach (Fig. 2) addresses these issues by providing a pattern-based framework for systematic application of formal methods in the V&V process. The basis of the proposed framework is a set of implementation-independent, configurable formal patterns for operating system constructs, such as tasks, resources, events, and alarms. Depending on the choice of system configuration, selected patterns are instantiated to generate a formal model of an operating system. We also define a set of constraint patterns that formally specify operational constraints of the operating system identified from the OSEK/VDX international standard. Given a formal OS model, test engineers can select a subset of constraint patterns with test options to generate a formal test environment of the operating system, from which rigorous test sequences can be auto-generated. Composed with automata representation of application software, the same OS model can be used for formal simulation/verification of the application software. We use model checker NuSMV (Cimatti et al., 1999) and SPIN (Holzmann, 1997) as back-

---

¹ A safety property states that no error state is reachable, where error states are typically identified using Fault Tree Analysis.
end computation engines for the simulation, formal verification, and test generation.

By pre-defining formal behavior patterns for the OS kernel and operational constraints, the proposed framework entirely eliminates the manual model construction process, saving effort and improving the quality of V&V.

3. Formalism for model generation

We define an embedded software as a synchronous composition of two types of state transition systems in general; one for the operating system generated from a set of configurations and corresponding OS patterns and the other for the application software synchronized with the operating system on a set of API function calls. This section introduces formal models for configurable operating systems and test generation.

Our formalism is based on parameterized statemachines (Shahbaz et al., 2007; Yavuz-Kahveci and Bultan, 2005) where a set of parameters are used to define transition relations.

Definition 1. \(M[C] = (S, S_0, V, E, P[C, V, E], A, R)\) is a parameterized statemachine whose transitions are defined by predicates over variables \(V\), parameters \(C\), or a set of events \(E\), where \(S\) is a set of system states, \(S_0 \subseteq S\) is a set of initial states, \(P[C, V, E]\) is a set of predicates over \(C, V\), and \(E\) is a set of actions, and \(R\) is a transition relation defined as \(R : S \times P[C, V, E] \times A \rightarrow S\).

A state in a statemachine consists of a system state and a valuation of the variables. A system state may transit to another system state while taking an action in \(A\) and changing the valuations of the variables, if a predicate over \(C, V\), and \(E\) is true. For notational convenience, a transition \(r \in R\) from \(s\) to \(t\) is represented as \(s \xrightarrow{p[C, V, E]/a} t\) for a predicate \(p\) over \(C, V\), and \(E\), and a sequence of actions \(a\). A transition may result in changes of system states and changes of variables through action statements.

An operating system is a synchronous parallel composition of parameterized statemachines.

Definition 2. An operating system for embedded software
\[
OS[C] = (S, S_0, V, E, P[C, V, E], A, R) = M_1[C_1] \parallel M_2[C_2] \parallel M_3[C_3] \parallel \ldots \parallel M_n[C_n]
\]
is a synchronous parallel composition of a set of parameterized state transition systems, where

1. \(S = S_1 \times S_2 \times \ldots \times S_n\), where \(S_i\) is the set of system states in \(M_i[C_i]\).
2. \(S_0 = S_1^0 \times S_2^0 \times \ldots \times S_n^0\), where \(S_i^0\) is the set of initial states in \(M_i[C_i]\).
3. \(C = \bigcup C_i\), i.e., the union of all parameters in each \(M_i\).
4. \(V = \bigcup V_i\), i.e., the union of all variables in each \(M_i\).
5. \(E = \bigcup E_i\), i.e., the union of all events in each \(M_i\).
6. \(A = \bigcup A_i\), i.e., the union of all actions in each \(M_i\).
7. \(R \subseteq R_1 \times R_2 \times \ldots \times R_n\) is a synchronized product of transitions in each \(M_i\), synchronized on \(P[C, V, E]\).

Synchronized parallel composition allows each statemachine to accept a transition while the others remain in the same state, but all statemachines satisfying transition conditions must take the transitions at the same time. The following shows operational semantics of the transition relations for a composition of arbitrary two statemachines \(M_i\) and \(M_j\), which can be easily extended to a composition of an arbitrary number of statemachines.

\[
\begin{align*}
&\left( S_i, S_j \right) \xrightarrow{p[C, V, E]/a} \left( t_i, t_j \right), \quad \left( S_i, S_j \right) \xrightarrow{q[C, V, E]/b} \left( t_j, t_j \right), \quad \left( S_i, S_j \right) \xrightarrow{q[C, V, E]/a} \left( t_i, t_j \right), \quad \left( S_i, S_j \right) \xrightarrow{q[C, V, E]/b} \left( t_j, t_i \right), \quad \left( S_i, S_j \right) \xrightarrow{q[C, V, E]/a} \left( t_i, t_j \right), \quad \left( S_i, S_j \right) \xrightarrow{q[C, V, E]/b} \left( t_j, t_i \right), \quad \left( S_i, S_j \right) \xrightarrow{q[C, V, E]/a} \left( t_i, t_j \right), \quad \left( S_i, S_j \right) \xrightarrow{q[C, V, E]/b} \left( t_j, t_i \right)
\end{align*}
\]

Definition 2 is for operating systems in general and does not contain specifics for defining an actual instance of an operating system. For example, the numbers and types of statemachines to be composed determine a specific operating system, as illustrated in Fig. 3. The system configuration determines the number of such objects and types, which is necessary information to generate an OS model. Therefore, the proposed model generator is parameterized in two dimensions, configuration values and the number of instantiations for each object type.

Definition 3. An operating system generator \(OS_g\) for embedded software is a function from configuration vectors to formal OS models.

\[
OS_g : C_{Tr}^n \times C_{Cr}^n \times C_{E}^n \times C_{A}^n \rightarrow OS
\]

where \(C_{Tr}, C_{Cr}, C_{E}\), and \(C_{A}\) represent a set of vectors of task configurations, resource configurations, event configurations, and alarm configurations, respectively. \(n, m, p, q\) and \(q\) are the numbers of instantiations of these objects.
For example, if a system configuration defines two tasks and two resources, the generated formal OS model $OS_{\text{g}}(c_1, c_2, r_1, r_2, \text{null}, \text{null}) = M_1(c_1) \times M_2(c_2) \times M_3(c_1) \times M_4(c_1) \times M_5$.

Definition 4. A configuration-dependent pattern for an embedded operating system is a pre-defined parameterized statemachine $M[c]$ with non-empty configuration $C$ for each operating system construct. It is configuration-independent if $C$ is empty.

A parameterized statemachine is pre-defined for each type $t$ of operating system constructs such that a formal OS model can be generated once a system configuration is determined. That is, a parameterized statemachine $M[c]$ is auto-generated for each configuration $c \in C_t$. Details of these patterns will be introduced in Section 4.

The formal OS model generated from the system configuration can be used (1) to auto-construct a test model for generating test sequences, (2) to validate the task design of the embedded software independently of the hardware platforms, and (3) to verify safe interaction behavior between the application and the operating system.

Definition 5. A test model generator $OS_{\text{g}}$ for an embedded operating system is a function from vectors of system configuration, constraint patterns, and coverage criteria to a set of formal test models.

$OS_{\text{g}} : C_1 \times C_2 \times C_3 \times C_4 \times C_{\text{cov}} \rightarrow M_T$.

where $C_i$ is a set of choices for constraint patterns and $C_{\text{cov}}$ is a set of coverage criteria.

Constraint patterns are a set of parameterized statemachines specifying constraints on operational environments of operating systems. The details of these patterns will be explained in Section 5. The coverage criteria determine the test generation strategy. Examples of coverage criteria include state coverage and transition coverage of the constraint patterns and/or OS patterns. This test model can be used for auto-generating test sequences for safety checking the implementations of an embedded operating system. The details of the test model will be explained in Section 6.

Finally, an embedded software is a composition of a configuration-dependent OS model and the statemachine of the application software.

Definition 6. An embedded software is a parallel composition of an embedded operating system $OS_{\text{g}}(C_T, C_R, C_E, C_A)$ and application software $M_{\text{app}}$ synchronized over the set of API function calls $E_{\text{API}}$.

$M_{\text{em}}[C_T, C_R, C_E, C_A] = OS_{\text{g}}[C_T, C_R, C_E, C_A] \parallel M_{\text{app}}$.

where

1. $OS_{\text{g}}(C_T, C_R, C_E, C_A) = (S, S_0, V, E_{\text{API}}, P[C, V, E_{\text{API}}], A, R)$, where $A$ is a set of value assignments to the variables in $V$.
2. $M_{\text{app}} = (S', S_0', V', E', A', R')$ is a statemachine, where $R' \subseteq S' \times P[V', E'] \times A' \times S'$ is a set of transition relations, and
3. $E_{\text{API}} \subseteq A'$, i.e., actions in the application software include the set of API function calls.

$M_{\text{em}}$ is a formal model for embedded software to check the validity of the application software with respect to its underlying operating system and required functional behaviors.

4. Parameterized statemachine patterns for OSEK/VDX operating systems

This section describes a set of pre-defined, parameterized patterns for embedded operating systems. These patterns are identified by analyzing the international standard for automotive ECU software, OSEK/VDX. Though OSEK/VDX is primarily for automotive control software, it is also adopted to define real-time operating systems for Robots and IoT systems Erika. Therefore, patterns introduced in this section are largely reusable for embedded operating systems in these domains with some extensions.

4.1. OSEK/VDX

OSEK/VDX is a joint project of the automotive industry to establish an industry standard for an open-ended architecture for distributed control units in vehicles. The OS part of OSEK/VDX has been adopted by major automobile manufacturers and by the AUTOSAR open-source architecture defined by a consortium of over 50 automotive manufacturers worldwide.

Fig. 3 illustrates our understanding of the structure of the OSEK OS. A set of Tasks, Resources, Events, and Alarms comprises a set of OSEK objects, and these objects comprise an OSEK operating system. Each OSEK object is distinguishable by its type and unique name within the OS kernel. The properties of OSEK objects are statically configurable.

AUTOSAR OS is an extension of OSEK OS, allowing multiple CPUs and facilitating convenient organization of task activation and

---

2 An ISR(Interrupt Service Routine) is considered as a special type of a Task.
event setting using schedule tables and the time frame of interrupt services. The following section introduces patterns of the basic constructs of OSEK OS. An extension to the AUTOSAR OS may require modeling of global timers to include schedule tables and the time frames of the interrupt services.

4.2. Patterns for OSEK/VDX operating systems

This section explains the functional requirements analyzed from the OSEK/VDX requirements specifications and corresponding patterns for essential constructs of the OS.

Throughout this section, we use $M^t$, $M^p$, $M^a$, $M^l$, and $M^g$ to represent statemachines for Tasks, Resources, Events, Alarms, and Schedulers, respectively. A member of a statemachine is represented using the ‘·’ symbol. For example, $M^t.S$ represents the set of state systems of a Task $M^t$ and $M^p.v$ represents a variable $v$ of a scheduler $M^p$. For notational convenience, we simply represent a statemachine with identifier id as $M(id)$, e.g., $M^t(tid)$.

4.2.1. Pattern for tasks

A task $\dagger$ is a basic building block of embedded control software, which interacts with the operating system kernel through system calls. Fig. 4 illustrates the state transition diagram of an extended-mode task. The state changes when the scheduler intervenes, invoked primarily by certain API function calls. For example, the state of a task transits from running to suspended when TerminateTask is called, and the task with the highest priority in the ready queue transits from ready to running.

When multiple tasks exist in a system, configuration factors determine the characteristics of an individual task; for example, a task must be distinguishable by an unique identifier, can be auto-started when the system starts, can wait for events if it owns them, or can be preempted. The pattern for tasks is based on the general behavior from the OSEK/VDX standard; however it is extended and generalized with parameters to consider the inter-relationship among multiple instances of the tasks and other objects.

**Definition 7.** A pattern for tasks $M^t(C_T) = (S, S_0, V, E, P[C_T, V, E], A, R)$, where

$C_T = \{ \langle tid, prior, auto, ext, preemptible, max_act_cnt, eidSet, ref(M^g.S), ref(M^h.S) \rangle \}$

$S = \{ \text{suspended, ready, running, waiting} \}$

$S_0 = \{ \text{suspended, ready} \}$

$V = \{ Dprior, WEvt, M^q.List, M^q.max_prio, M^q.run_tid, M^g.cell_prio \}$

$E: \text{a set of API function calls defined in the OSEK/VDX standard}$

$A: \text{a set of value assignment to the variables in } V, \text{ and}$

$R \subseteq S \times P[C_T, V, E] \times A \times S$

$C_T$ is a set of configuration vectors consisting of system configuration for the task module and internal parameters for referencing system states of other modules; $tid$ is a task identifier, $prior$ is the priority of the task, and $auto$, $ext$, and $preemptible$ are Boolean values specifying whether the task is auto-starting, is an extended task, and is preemptible, respectively. $max_act_cnt$ is the maximum number of allowed task activations\(^4\) and $eidSet$ is the set of event identifiers owned by the task. All these values must be determined to construct an operating system. The types of internal parameters are fixed to references to $M^g.S$ and $M^h.S$, but the number of internal parameters is determined by system configuration, depending on the number of resources and events in the system.

Except for $C_T$, the others are fixed in the pattern. The statemachine refers to its own local variables, $Dprior$, $WEvt$, and the local variables of $M^q$ and $M^g$. $Dprior$ is defined for the dynamic priority of the task, which is set equal to $prior$ in the initial state. $WEvt$ is for the event identifier for which the task is waiting.

OSEK/VDX requirements specify 26 API functions. We regard a call to an API function as an event in the system and define $E$ as a set of API function calls. $R$ includes seven transitions as illustrated in Fig. 5 illustrating only the predicates for each transition. Examples of complete transitions are:

$$r_2: \text{ready} \xrightarrow{\text{runtime}} \text{running}, \text{ where } p = (M^g.run_tid \leq 0) \text{APICall} = \text{Schedule}$$

$$\text{APICall} = \text{SetEvent} \text{APICall} = \text{ReleaseResource} \&$$

$$(\text{GetFirstElem}(M^2.List) = \text{tid} \& \text{prior} > M^2.max.prio), \text{ and}$$

$act = (M^g.run_tid := \text{tid}; \text{RemoveFirst}(M^2.List); Dprior := prior)$$

$$r_6: \text{suspended} \xrightarrow{\text{sched}} \text{ready}, \text{ where } p = (\text{APICall} = \text{ActivateTask}(id)$$

$$\& \text{tid} = id), \text{ and } act = \{ \text{Insert}(M^2.List, < \text{tid}, \text{prio} >); \}$$

For example, transition $r_2$ is enabled when $M^g(tid)$ is in the ready state on the condition that there is no other running task in the system or the API call is one of Schedule, SetEvent, and ReleaseResource, and the task is the first task in the ready list with the highest priority among all the tasks ready to run. As the result of the transition, the state of $M^g(tid)$ changes to running, the identifier of the currently running task becomes tid, the task is removed from the list of tasks in ready state, and the dynamic priority $Dprior$ is set to the initial static priority $prior$ of the task.

Any number of tasks can be instantiated using this pattern by setting the values for $C_T$.

4.2.2. Patterns for resources, events, and alarms

The OSEK priority ceiling protocol is designed to avoid the problems of priority inversion and deadlocks. The priority of each task and ceiling priority of each resource are assigned statically, and when a task requires a resource with a higher ceiling priority, the priority of the task is promoted to the ceiling priority of the resource. All resources have their own unique identifiers and ceiling priorities, but share common behaviors as defined in the following pattern.

**Definition 8.** $M^g(C_R) = (S, S_0, V, E, P[C_R, V, E], A, R)$ is a pattern for resources, where

$C_R = \{ \langle rid, ceil.prio > \}$

$S = \{ \text{free, held} \}$

$S_0 = \{ \text{free} \}$

$V = \{ \text{prev_prio}, M^g.Dprior, M^g.run_tid \}$

$R \subseteq S \times P[C_R, V, E] \times A \times S$

Fig. 6(a) illustrates the pattern for resources. This pattern refers to the $Dprior$ of the tasks and $run_tid$ value of the scheduler. For

\(^{\dagger}\) A thread is another name of a task in real-time operating systems.

\(^{4}\) OSEK/VDX requires setting a maximum number of activations per task in the system configuration, which cannot be changed at runtime.
example, if GetResource(id) is called when the resource with identifier id is in the free state, it transits to held state and the local variable prev_prio is updated with the dynamic priority of the currently running task (r1). If the ceiling priority of the resource is greater than the dynamic priority of the calling task, the dynamic priority of the task gets promoted to the ceiling priority of the resource (r1). r1 in Fig. 6 (a) shows the common part of these two transitions.

\[ r1 : \text{free} \overset{\text{GetResource}}{\rightarrow} \text{held}, \; \text{where} \; p = \{(\text{APICall} = \text{GetResource}(id)) \land \text{rid} = \text{id} \land \text{cell_prio} \leq M^r(\text{run_tid}).\text{Dprio}, \text{and} \; \text{act} = \{\text{prev_prio} := M^r(\text{run_tid}).\text{Dprio})\}. \]

According to the OSEK/VDX standard, events are not independent objects; Rather, they are assigned to extended tasks. Each extended task has a definite number of events. Therefore, our pattern for events uniquely defines an event with its own identifier and the identifier of its own task. Events do not maintain local variables in the pattern; Setting or waiting events causes a rescheduling of the tasks, as specified in the pattern for the tasks; No other actions are defined except for state transitions in the event pattern (6 (b)).

Definition 9. \( M^E(C_E) = (S, S_0, V, E, P[C_E, V, E], \emptyset, R) \) is a pattern for events, where

\[ C_E = \{< \text{eid}, \text{tid} >\}, \; S = \{\text{set}, \text{clear}\}, \; S_0 = \{\text{clear}\}, \]

\[ V = \{M^r(\text{tid}).\text{WEvt}\}, \; \text{and} \; R = \{(r_0, r_1, r_2) \subset S \times P[C_E, V, E] \times S\} \]

The OSEK operating system provides a special mechanism alarm for processing recurring events, such as timers and periodic tasks. When an alarm expires, the operating system can activate tasks, set events, or call an user-defined application function named alarm-callback routine.

Definition 10. \( M^A(C_A) = (S, S_0, V, E, P[C_A, V, E], A, R) \) is a pattern for alarms, where

\[ C_A = \{< \text{aid}, \text{time} >\}, \; S = \{\text{expired}, \text{set}, \text{alarmed}\} \] and \( S_0 = \{\text{expired}\}, \]

\[ V = \{\text{counter}, \text{periodic}\}, \; \text{and} \]

\[ R = \{(r_0, r_1, r_2, r_3, r_4) \subset S \times P[C_A, V, E] \times S\} \]

An alarm is identified by its name and time limit. Fig. 7 illustrates the behavior of an alarm defined in this pattern. Unlike other patterns, there could be a user-defined actions, APICallSeq, which may not be pre-defined or identified from the system configuration information. Our pattern defines this as an initially empty set of actions which can be replaced with user-defined function calls later, when an instance of the operating system is composed with an application software. Though it is omitted from the figure, ClearAlarm function call makes the machine transits to the expired states no matter which state it was in.
4.2.3. Pattern for the scheduler

An operating system maintains a scheduler, a FIFO scheduler according to OSEK/VDX standard. We assume one and only one scheduler for an operating system and its behavior is configuration- and implementation-independent. Therefore, any operating system instance includes an instance of the following pattern.

Definition 11. $M^O = (S_0, S, V, E, P[V, E], A, R)$ is a pattern for a FIFO scheduler, where

$S = \{\text{empty}, \text{some}, \text{DError}, \text{SError}\}$ and $S_0 = \{\text{empty}\}$,

$V = \{\text{act_cnt}[n], \text{max_prio}, \text{List, run_tid}, M^f (id), Dprior, M^f (id), \text{max_act_cnt}\}$ and

$R \subset S \times P[V, E] \times A \times S$.

A scheduler maintains an activation count for each task, the largest value of the priorities of the activated tasks, a list of activated tasks in the descending order of priorities, and the identifier of the currently running task. It also refers to priorities and the maximum number of activation count for each task.

Fig. 8 illustrates the state transition relation of the scheduler;

For example, the scheduler transits from empty to some (transition $r_1$) when the ActivateTask API function is called and the activation count of the activated task is less than its maximum number of activation count. The result of the transition is an increment of the activation count and an insertion of the activated task into the list of activated tasks, which is then sorted in the descending order of priorities. $r_3$ is the transition for the scheduling point when Schedule or ReleaseResource is called; If the dynamic priority of the currently running task is less than the priority of the task in the ready list, the scheduler removes the first element of the list, which becomes the next running task and inserts the currently running task into the ready list. The transitions $r_4$ and $r_5'$ treat similar cases: When TerminateTask is called, the scheduler removes the first task in the ready list which becomes the next running task and decreases the activation count of the currently running task. It transits to some($r_4$) or empty($r_5'$) depending on the size of the ready list. The case when ChainTask is called is omitted from the figure to save the space, as its behavior is the same as calling ActivateTask and TerminateTask in the order.

Table 1 summarizes the patterns $M^I, M^D, M^G, M^H, M^O$ with respect to their parameters, system states, variables, and related events. $C$ is the set of parameters whose values are defined by system configuration. $V$ is the set of local variables as well as the variables of other patterns to be referenced. The local variables are fixed in the pattern, however, the references to the variables and states of other patterns may change depending on the number of instances of the other patterns in the system. For example, $M^I$ refers to $M^O(tid, eid).S$ for any $tid$ and $eid$ pair and the number of such $M^O$s is determined by the system configuration. Conversely, $S$ and $E$ are fixed for each pattern and do not change as long as the OSEK/VDX standard remains the same.

4.3. Validity of the patterns

We implemented an OS model generator that generates formal OS models in NuSMV using the OS patterns defined in this section. The validity of the OS patterns were verified using LTL model checking using NuSMV against a list of the functional properties of operating systems identified from the OSEK/VDX international standard. Model checker NuSMV verifies that the generated formal OS models satisfy the same list of functional properties required by the standard or generates counterexamples if properties are not satisfied. Our patterns are the result of two years of continuous validation and revision.

5. Constraint patterns

Patterns systematize the model generation for configurable embedded operating systems. The generated formal model can be used to verify safety properties of the operating system before it is actually implemented or to generate test sequences for final implementation code. However, both verification and test generation require understanding of its operational environment, event sequences in the case of OSEK/VDX operating systems. A typical manner of modeling a complete environment is to include the
set of arbitrary sequences of API function calls; However, it is extremely costly and inefficient to perform V&V under such an environment. To systematically abstract the operational environment while maintaining comprehensiveness, we suggested a set of patterns, named constraint patterns (Choi, 2013; Choi and Byun, 2017; Byun and Choi, 2015). Defined from operational constraints in the OSEK/VDX standard, the patterns are the tool to systematically partition the complete environment and construct an abstract environment.

Because constraint patterns have an important role in the proposed framework, this section briefly summarizes those patterns in three categories, sequence constraints, configuration-dependent constraints, and state-dependent constraints. The content of this section is borrowed from Choi (2013) and Byun and Choi (2015).

5.1. Sequence constraint patterns

Sequence constraints restrict the possible sequences of API function calls; Some examples include: certain system calls that must be followed by matching system calls, limiting the number of calls to a certain API function, or restricting a specific API function from being called between two specific system calls. The sequence constraint patterns capture and generalize such constraints:

**Definition 12.** (Sequence constraint patterns) Let $Σ$ be a set of API functions and $N$ be a set of natural numbers. For any $f, f_1, f_2 \in Σ$, $A \subseteq Σ$, and $n \in N$,

1. $\text{InPairs}(f_1, f_2)$: $f_1$ and $f_2$ shall be called in pairs in the order of $f_1$ followed by (not necessarily directly) $f_2$.
2. $\text{Limited}(f, n)$: The number of calls to $f$ shall not exceed $n$.
3. $\text{SetLimited}(A, n)$: The total number of calls to the functions in $A$ shall not exceed $n$.
4. $\text{NotInBetween}(f_1, f_2)$: A call to $f$ shall not be allowed in between calls to $f_1$ and $f_2$.
5. $\text{MustEndWith}(A)$: $f \in A$ shall be called eventually and no call shall be allowed afterward.

These patterns are formalized using pushdown automata (Choi and Byun, 2017). Fig. 9 is an illustration of a simplified version of the automata using counters instead of stacks. $f, f_1, f_2$ and $A$ represent the formal parameters defined in each constraint pattern, and $n$ is a counter variable which is initialized to zero at the start. States other than the acceptance state denote error states that imply the constraint is violated. When a sequence of API functions is given as an input, the state of each automaton transits according to the sequence and reports whether the sequence satisfies or violates the constraint. The formal definitions of sequence constraint patterns can be found in Choi and Byun (2017).

For example, consider a constraint specified as $\text{InPairs(\text{GetResource,ReleaseResource})}$, which claims that the two APIs should appear (not necessarily directly) in pairs. Then a constraint automaton is instantiated, with $f_1$ and $f_2$ corresponding to GetResource and

---

**Table 1 Patterns for operating system constructs.**

<table>
<thead>
<tr>
<th></th>
<th>$M^0$</th>
<th>$M^1$</th>
<th>$M^2$</th>
<th>$M^3$</th>
<th>$M^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>tid, prio, auto, ext, eidSet, max_act_cnt, preemptible, ($M^0$, $M^0$.s)</td>
<td>rid, ceil_prio</td>
<td>eid, tid</td>
<td>aid, time</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>suspended, ready, running, waiting</td>
<td>free, held</td>
<td>set, clear</td>
<td>expired, set, alarmed</td>
<td>empty, expired, empty</td>
</tr>
<tr>
<td>$S_0$</td>
<td>free</td>
<td>clear</td>
<td>expired</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>$V$</td>
<td>$D_{privo, WEvt, M^0.cell_prio, M^0.List, M^0.max_prio, M^0.run_tid}$</td>
<td>$M^0$.WEvt</td>
<td>counter, periodic</td>
<td>act_cnt[n], $M^0$.Dprivo, $M^0$.max_act_cnt, max_prio, run_tid, List</td>
<td></td>
</tr>
</tbody>
</table>

* $C$: parameters, $S$: system states, $S_0$: initial states, $V$: variables, $E$: events

**$M^0$:** task, $M^0$: resource, $M^0$: event, $M^0$: alarm, $M^0$: scheduler.

---

**Fig. 9.** Constraint automata for sequence constraints.
ReleaseResource, respectively. When there is a sequence of API calls 
\text{GetResource}(r1) - \text{GetResource}(r2) - 
\text{ReleaseResource}(r2) - \text{ReleaseResource}(r1) - 
\text{Schedule}() - \text{ReleaseResource}(r1), the constraint automaton 
transit is as follows: \(s_0(n = 0) \rightarrow s_1(n = 1) \rightarrow s_2(n = 2) \rightarrow s_3(n = 1) \rightarrow s_4(n = 0) \rightarrow s_5\), 
terminating in a non-terminal state, and, thus, violating the 
\text{InPairs}(\text{GetResource}, \text{ReleaseResource}) constraint.

It is straightforward to represent these patterns in parameterized 
state machines as defined in \textbf{Definition 1}, as we can define 
\(C = \{f_1, f_2, A, n\} \in \{s_0, s_1, s_2\}\) (the same as the states in the 
constraint automaton), \(V = [n, \max] \in \{\Sigma\}. A = [n +, n -],\) and 
P:\(C.V.E = [n == 1, n \leq \max, n > \max, n > 1]\).

5.2. Configuration-dependent constraint patterns

Configuration-dependent constraint patterns specify 
configuration-dependent aspects of API function calls, such as 
the statically assigned priority of tasks or statically configured 
mode of tasks used as parameters in function calls. Examples are:

CC 1 A task should not attempt to acquire a resource when 
the statically assigned priority of the calling task is higher than 
the calculated ceiling priority.

CC 2 When a task calls the \text{SetEvent} API function, the referenced 
task shall be an extended task.

To address the configuration-dependent aspects of API function 
calls, the set of API functions \(\Sigma\) is categorized into three groups 
\(\Sigma_T, \Sigma_R, \Sigma_E \in \Sigma\), according to the type of OSEK objects to which 
they refer as a parameter. For example, an API function in \(\Sigma_T = 
\{\text{ActivateTask}, \text{ChainTask}, \text{SetEvent}\}\) passes a parameter of Task type, 
an API function in \(\Sigma_R = \{\text{GetResource}, \text{ReleaseResource}\}\) passes a pa-
rameter of Resource type, and an API function in \(\Sigma_E = \{\text{WaitEvent}, \text{ClearEvent}\}\) 
passes a parameter of Event type.

\textbf{Definition 13.} (Configuration-dependent constraint patterns) 
Given an API function \(f \in \Sigma\), a subset of APIs \(A \subseteq \Sigma\), type \(t \in \{\text{Task, ISR}\}\), 
an arbitrary system call \(\sigma = (\tau, f_r, p_{\sigma_r})\).

1. \text{CallerMode}(f, m) \overset{\text{def}}{=} (f_r = f \rightarrow \tau_r, \text{mode} = m) : \text{If the API func-
tion of a system call} \sigma \text{is equal to} f, \text{then the mode of the task calling} \sigma \text{shall be} m.

2. \text{CallerMode}(f, m) \overset{\text{def}}{=} (f_r = f \rightarrow p_{f_r}, \text{mode} = m) : \text{If the API of a system call} \sigma \text{is equal to} f \in \Sigma, \text{then the mode of the task referenced by} f \text{shall be} m.

3. \text{CallerType}(A, t) \overset{\text{def}}{=} (f_r \in A \rightarrow \tau_r, \text{type} = t) : \text{If the API of a sys-
tem call} \sigma \text{is in a set} A, \text{then the type of the caller shall be} t.

4. \text{OwnerOnly}(f) \overset{\text{def}}{=} (f_r = f \rightarrow \tau_r, \text{owner} p_{\sigma_r}) : \text{If the API of a system call} \sigma \text{is equal to} f \in \Sigma, \text{then the task that calls the API shall own the event referenced by} f.

5. \text{CeilingPriority}(f) \overset{\text{def}}{=} (f_r = f \rightarrow p_{f_r}, \text{priority} \geq \tau_r, \text{priority}) : \text{If the API of a system call} \sigma \text{is equal to} f \in \Sigma, \text{then the ceiling priority of the resource} p_{f_r} \text{referenced by} f \text{shall be greater than or equal to the statically assigned priority of the calling task} \tau_r.

Each constraint pattern is defined as a function that 
returns true when the condition is satisfied 
for each system call. For example, a constraint instance 
\text{CallerMode} \text{(WaitEvent, extended)} returns true 
for a system call in a system where the caller is an extended task 
owning one event. However, \text{OwnerOnly}(WaitEvent) may 
not be true if the caller does not own the event specified in the parameter.

CC 1 can be specified as \text{CeilingPriority} \text{(GetResource)}. 
Likewise, CC 2 can be expressed as \text{CallerMode}(\text{SetEvent}, 
\text{extended}) using these patterns.

5.3. State-dependent constraint patterns

Other constraints may require consideration of internal states 
of a system. An example of such constraints is:

SC 1 If the referenced task is in the suspended state, the events 
cannot be set.

\textbf{Definition 14.} (State-dependent constraint patterns) For an arbitrary 
of \(f \in \Sigma,\) a set of task states \(S = \{\text{suspended, ready, running, waiting}\},\) 
a resource state \(s' \in \{\text{held, free}\},\) and an event state \(s'' \in \{\text{clear, set}\},\)

1. \text{CeilCallState}(f, S) \overset{\text{def}}{=} (f_r = f \rightarrow p_{f_r}, \text{state} \in S) : \text{If the API of a system call} \sigma \text{is equal to} f \in \Sigma, \text{then the state of the task referenced by} f \text{shall be in} S.

2. \text{ResCallState}(f, s') \overset{\text{def}}{=} (f_r = f \rightarrow p_{f_r}, \text{state} = s') : \text{If the API of a system call} \sigma \text{is equal to} f \in \Sigma, \text{then the state of the resource referenced by} f \text{shall be equal to} s'.

3. \text{EventState}(s'') \overset{\text{def}}{=} (f_r = f \rightarrow p_{f_r}, \text{state} = s'') : \text{If the API of a system call} \sigma \text{is equal to} f \in \Sigma, \text{then the state of the event referenced by} f \text{shall be equal to} s''.

For example, SC 1 can be specified as \text{CeilCallState} 
(\text{SetEvent}, \{\text{ready, running, waiting}\}).

Unlike sequence constraint patterns, configuration-dependent 
or state-dependent constraint patterns are defined as Boolean 
functions, defining parameterized predicates over API function 
calls. The state machine for these constraint patterns is in a simple 
form with two states, one for the system that satisfies the given 
constraint and the other for the system that dissatisfies the 
constraint. \text{Fig. 10 (a)} illustrates a template for the constraint patterns. 
\text{Fig. 10 (b)} is an instantiation of the template.

Similarly to the sequence constraints, configuration- and 
state-dependent constraint patterns can also be represented as 
parameterized state machines, where \(C\) is a set of API functions and 
their parameters and \(P(C, V, E)\) is the set of constraints, as the truth 
value of each constraint determines state transitions.

6. Pattern-based verification and validation

As described in the previous two sections, the basic constructs 
of OSEK/VDX operating systems are formalized as parameterized 
and configurable state machines and the operational environ-
ment of the operating system is abstracted as a set of oper-
atinal constraint patterns. The proposed V&V framework allows 
engineers to select appropriate functional patterns and constraint 
patterns based on desired system configurations. Depending on 
the choices, various operating systems, together with various environment 
models, can be auto-generated for verification and validation 
purposes.

For example, Fig. 11 presents an implementation of the frame-
work; the set of pre-defined patterns are specified in modeling 
languages Promela (Holzmann, 2003) and NuSMV. The patterns 
specified in NuSMV are used to construct test models that act as
basis for test sequence generation for testing all possible combinations of constraint satisfaction. It is also used to check whether a
given control software complies API call constraints. The patterns
specified in Promela are used to construct a simulation model
by composing an instance of an operating system model with a
state machine representation of the application code.

Any formal languages can be used to specify these patterns,
however, we chose these languages because they are the input lan-
guages of the most widely used model checkers, and therefore,
convenient for performing various V&V activities.

Among various possible applications of the proposed frame-
work, this section explains test generation for safety checking OS
kernel implementations in more detail because it requires both OS
patterns and constraint patterns, and, thus, is an excellent example
to demonstrate how the suggested patterns can be utilized. How
this framework is utilized for API-call constraint checking can be
found in Kim et al. (2016).

6.1. Patterns and test models

A test model is constructed by assembling a set of OS patterns
and constraint patterns depending on the system configurations
and test options. We follow the general framework of specifi-
cation-based test generations using model checking (Offutt et al.,
2003; Stocks and Carrington, 1996). The major differences, however, are
that the proposed approach generates formal models from param-
erized patterns and guides test generation using constraint be-
havior and functional behavior.

Fig. 11 is an example of a test model assembled from the set of
patterns for a given system configuration and test options. The top
left part of the figure displays a part of the system configuration in
the Oil (Feiler, 2003) format; There are two tasks (t1, t2), two
resources (r0, r2), and one event (e0) owned by the task t2. The
upper right part of the figure (surrounded by dashed lines) is a set
of patterns used to model the operating system of the configura-
tion, including two instantiations of the task pattern, two instantia-
tions of the resource pattern, an instantiation of the event pattern,
and a scheduler. The lower left part of the figure displays a set of
constraints that could be imposed on the model using constraint
patterns. The right lower part of the figure (surrounded by dotted
lines) is the set of constraint automata representing the imposed
constraints. Then, the test model OS_{T} is

\[
OS_{T} = M^{t}(r1) || M^{t}(t2) || M^{r}(r0) || M^{e}(e0) || M^{C2} ||
C^{1} || C^{2} || C^{3} || C^{4} || C^{5} || C^{6} || C^{7}.
\]

The test model does not constrain the environment of the oper-
ating system as it is; Rather, it monitors whether a sequence of
API calls violates any constraints. For example, Table 2 indicates
the change of system states for \( M^{t}(r1), M^{t}(t2), M^{r}(r0), C^{2}, \) and
\( C^{3} \) when \( r1 \) calls GetResource(r0), TerminateTask(),
ReleaseResource(r0) in the order, assuming that the system just
starts up and \( r1 \) is running (since it is an autostart task). The
constraint automata \( C^{2} \) and \( C^{3} \) reach to non-terminal states in
the third and fourth API call, respectively, meaning that the system
violates these two constraints under this call scenario.

<table>
<thead>
<tr>
<th>API calls</th>
<th>( M^{t}(r1) )</th>
<th>( M^{t}(t2) )</th>
<th>( M^{r}(r0) )</th>
<th>( C^{2} )</th>
<th>( C^{3} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 StartOS</td>
<td>running</td>
<td>suspended</td>
<td>free</td>
<td>( s_{0} )</td>
<td>( s_{0} )</td>
</tr>
<tr>
<td>2 GetResource</td>
<td>running</td>
<td>suspended</td>
<td>held</td>
<td>( s_{1} )</td>
<td>( s_{0} )</td>
</tr>
<tr>
<td>3 TerminateTask</td>
<td>running</td>
<td>suspended</td>
<td>held</td>
<td>( s_{2} )</td>
<td>( s_{1} )</td>
</tr>
<tr>
<td>4 ReleaseResource</td>
<td>running</td>
<td>suspended</td>
<td></td>
<td>( s_{2} )</td>
<td>( s_{2} )</td>
</tr>
</tbody>
</table>

Table 2
An API call sequence and corresponding change of states.
When we aim at the verification of code safety for a given operating system, the test model can be used to identify at least two API call sequences for each constraint—one that satisfies the constraint and the other that does not, using the counterexample generation capability of model checking.

6.2. Trap properties and test generation

To elicit the desired test sequences from the test model, a set of falsifiable verification properties, or trap properties (Tan et al., 2004) are designed. The test generator provides two basic coverage criteria, state coverage and transition coverage. For state coverage, a trap property is asserted for a given state vector \(<s_j>\) in the test model that "there is no path from the initial state to \(<s_j>\). For transition coverage, a trap property is asserted for a given pair of state vectors \(<s_i>, \ldots, <s_j>\) in the model that "there is no transition from \(<s_i>\) to \(<s_j>\). These trap properties are auto-generated from the test model depending on the chosen coverage criteria. The model checker NuSMV is used to find a counterexample, i.e., a path that covers the specified state or transition, if there is any.

**Definition 15** (Trap properties). Let \(M = M_1 \mid M_2 \mid \ldots \mid M_n\) be a synchronized parallel composition of state machines, where the set of states for each \(M_i\) is denoted as \(S_i\).

1. Trap properties for state coverage: for any state \(<p_1, p_2, \ldots, p_n>\) and the initial state \(<s_0^1, s_0^2, \ldots, s_0^n>\) in \(S_1 \times S_2 \times \cdots \times S_n\),

\[tp^S(<s_1, s_2, \ldots, s_n>) \overset{def}{=} G(\bigwedge_{i=1}^n S_i = s_0^i) \rightarrow F(\bigwedge_{i=1}^n S_i = p_i)\]

2. Trap properties for transition coverage: for any states \(<p_1, p_2, \ldots, p_n>\) and \(<q_1, q_2, \ldots, q_n>\) in \(S_1 \times S_2 \times \cdots \times S_n\),

\[tp^T(<p_1, p_2, \ldots, p_n>, <q_1, q_2, \ldots, q_n>) \overset{def}{=} G(\bigwedge_{i=1}^n S_i = p_i) \rightarrow F(\bigwedge_{i=1}^n S_i = q_i)\]

The trap properties are expressed using Linear-time Temporal Logic with an additional abbreviated symbol representing the conjunctions of Boolean expressions; \(G\) and \(F\) are temporal operators meaning "always" and "in the future", respectively. \(\bigwedge_{i=1}^n S_i = s_0^i\) is an abbreviation for \(S_1 = s_0^1 \land S_2 = s_0^2 \land \ldots \land S_n = s_0^n\). Example of trap properties for the test model OS2 in Fig. 12 are:

\[tp^S(\ldots) = G((c_1^1 \land c_2^2 = s_0^1 \land c_3^3 = s_0^2 \land c_4^4 = s_0^5) \rightarrow C_5^5 = s_0^6 \land C_6^6 = s_0^7 \land (C_7^7 = s_0^8 \lor C_8^8 = s_0^9))\]

\[tp^T(\ldots) = G((c_1^1 = s_1^1 \land c_2^2 = s_1^2 \land c_3^3 = s_1^3 \land c_4^4 = s_1^4 \land c_5^5 = s_0^5 \land c_6^6 = s_0^7 \land C_7^7 = s_0^8 \land C_8^8 = s_0^9) \rightarrow F(c_3^3 = s_1^3 \land c_4^4 = s_1^4 \land c_5^5 = s_1^5 \land C_6^6 = s_1^6 \land C_7^7 = s_1^7 \land C_8^8 = s_1^8))\]

One possible counterexample for \(tp^S\) above is that the task \(t_1\) calls \(GetResource(0)\) and then calls \(ClearEvent(\epsilon)\). The
call GetResource causes \( C^1, C^2, \) and \( C^3 \) to transit to \( S_1 \) from \( S_0 \), whereas the others remain the same. Because the task \( t_1 \) is not an extended task, the call to API function ClearEvent makes the automaton \( C^5 \) transit from \( S_0 \) to \( S_1 \) afterwards. A model checker exhaustively searches the reachable set of states to determine a path reachable to the target state from the initial state.

### 7. Experiments

We have developed a toolset for test generation, task simulation, and API-call constraint checking using the suggested pattern-based V&V framework. The test generation tool is specialized for safety checking OS kernel code,\(^6\) the task simulation tool is for validating control logic of application software (Park et al., 2016), and the API-call constraint checking tool is for checking whether a specific control software complies API call constraints (Kim et al., 2016).

Focusing on the test generation part of the toolset, this section reports the evaluation result of the pattern-based V&V framework with respect to the time for the test sequence generation and the quality of the auto-generated test sequences compared to (1) those manually generated by domain experts, and (2) those auto-generated by using the state-of-the-art test input generation technique, concolic testing.\(^7\)

#### 7.1. Goal of testing

For focused experiments, we set the goal of testing to identifying failure-causing faults from an operating system under arbitrary interaction with application programs.

Fig. 13 is a motivating example from Trampoline (Beachennec et al., 2006) illustrating a fragment of an actual kernel code compliant to the OSEK/VDX standard. It displays a part of the variable declarations and the internal kernel function corresponding to the SetEvent for an OS with two tasks \( t_0 \) and \( t_1 \), where \( t_0 \) owns the events and \( t_1 \) does not. If \( t_1 \) calls SetEvent \((t_1, e_0)\) with \( task_id = 1 \), it accesses \( 
\text{tpl\_task\_events\_table}[1] \)

in line 6 and \( \text{events} \rightarrow \text{evt\_set} \)

in line 8 while \( \text{events} \) is null, causing either an array bound error or a null-pointer de-referencing error depending on the compiler used. This is a typical cause of system crash resulted from unsafe software.

Identifying such subtle issues and corner cases are known to be difficult and expensive as it often requires comprehensive verification. Nevertheless, a systematic approach using the proposed pattern-based V&V framework may improve the situation. The failure case in Figure 13, for example, could be identified by generating a test sequence violating the configuration-dependent constraint pattern \( \text{CallerMode(SetEvent, extended)} \) meaning that the caller of the SetEvent must be an extended task that owns events.

#### 7.2. Cost for test generation

We performed a series of experiments using the following configurations and coverage criteria to check the cost of pattern-based test generation.

1. Configurations: Two elements of configurations, the number of tasks and the number of constraints, were increased from two to four, while the others are fixed to two.

2. Coverage: Four types of coverage criteria were used as follows:

   a) Constraint state coverage \((CS)\): To cover each state in the constraint automata;

   b) Constraint state coverage, all combinations \((CS_{\text{all}})\): To cover all combinations of states in the constraint automata;

   c) Task state coverage \((TS)\): To cover all task states in the OS model.

   d) Task state coverage, all combinations \((TS_{\text{all}})\): To cover all combinations of task states in the OS model.

Table 3 illustrates the performance of the test generation. The first column from the left displays the system configuration in terms of the number of tasks and the number of constraints, while the others are fixed. From the second column, each pair of columns indicates the time for generating the test sequences (in seconds) and the number of test sequences generated, respectively, for the coverage option \( CS, CS_{\text{all}}, CS \times TS, \) and \( CS \times TS_{\text{all}} \) in that sequence. The number of generated test sequences is represented by \#\( P(\#S)\), where \#\( P \) represents the number of trap properties generated and \#\( S \) is the final number of test sequences (after eliminating duplications) generated from model checking those trap properties.

As we used model checking to generate counterexamples from the trap properties, it was expected that the time for test generation would increase exponentially as the numbers of tasks and constraints increased. Table 3 confirms our expectation, but within reasonable cost; it requires less than two seconds to model check one trap property in the worst case. The amount of memory required was within 25 Mbytes because we used the bounded model checking provided by NuSMV with search depth of 20 to 30. The number of test sequences generated was considerably less than the number of trap properties because many of the target states in the trap properties were not reachable. For example, the tool generated 4864 trap properties for the test model with four tasks and four constraints using \( CS \times TS_{\text{all}} \) coverage; However, only 1152 test sequences were generated from these properties.

All the experiments were performed on a Fedora machine with an Intel Xeon 3.4GHz e3-1270 processor and 32GB of 1333MHz DDR3 RAM.

#### 7.3. Failure cases

The test sequences generated from the tool were automatically converted into test drivers that were used to test Trampoline (Beachennec et al., 2006), an open-source implementation of the OSEK/VDX operating system. Trampoline supports the POSIX environment and has high source code quality as it was previously certified as OSEK/VDX compliant. Four failure cases were detected.

---

\(^{6}\) A video clip is available at [http://sselab.dothome.co.kr/wordpress/index.php/tool-demo-nusek](http://sselab.dothome.co.kr/wordpress/index.php/tool-demo-nusek) for the introduction of the test generation tool.

\(^{7}\) The result of concolic testing is an excerpt from the reference (Byun and Choi, 2015).

---

![](Fig. 13. A unsafe code fragment leading to system failure.)

```c
1. tpl_task_events t0_task_evt = {0, 0};
2. tpl_task_events * const tpl_task_events_table[1] = { &t0_task_evt };
3. ...
4. tpl_status tpl_set_event(task_id task_id, tpl_event_mask incoming_event){
5. ...
6. tpl_task_events * const events = tpl_task_events_table[task_id];
7. if(task->state != (tpl_proc_state)0x0){
8. events->evt_set |= incoming_event;
9. ...
10. }
```
while testing Trampoline in standard mode using generated test sequences;

- **V.1:** An assertion violation and a system failure captured by a violation of InPairs(GetResource, ReleaseResource) followed by a system call that involved rescheduling.
- **V.2:** A system failure captured by a violation of CallerMode(WaitEvent, extended).
- **V.3:** A system failure caused by setting an event to a non-extended task, violating CalleeMode(SetEvent, extended).
- **V.4:** An array bound error captured by CallerMode(ClearEvent, extended).

These failure cases were caused by subtle software issues that are difficult to identify. For example, V.4 was caused by an inconsistency between the system configuration and application program code. In Trampoline, events are managed as an array of integers for each task; when there are two extended tasks in a system of three tasks, the size of the array is statically set to two, if one task out of three does not own any events. However, when ClearEvent was called from a non-extended task, it attempted to access an array of events with an index out of bounds, resulting in a segmentation fault. This case was detected from a test sequence that violated CallerMode(ClearEvent, extended).

The four failure cases were identified with 65 test sequences generated from the configuration (3,4) and CS × TS coverage criteria.

### 7.4. Effectiveness of tests

To demonstrate the effectiveness of the proposed pattern-based test sequence generation, we compared the quality of the test sequences generated from our tool to that of a conformance test suite from an international certification agency that was manually constructed, but highly optimized by domain experts.

The conformance test suite contained 164 test cases for a maximum of five tasks, three events, and seven resources, where each test case consisted of a system call sequence varying in length from five to 100. The purpose of the test suite was to confirm that an OS implementation complies with OSEK/VDX requirements, mainly focusing on functional correctness. Each test case performs a sequence of API function calls and checks whether the system produces the expected output, e.g., an expected state of a task or an expected error code. Each system call in the test case was supported by a function to check (1) whether the system calls were executed in an expected sequence, (2) whether an appropriate error code was returned for each system call, (3) whether the states of the tasks transitioned as expected by each system call that caused rescheduling, and (4) whether the functional correctness of the hardware-dependent parts such as interrupt service routine and alarm were tested by invoking them from the hardware.

Comparing both test suites from the certification agency and from the proposed tool revealed the following strength and weakness with respect to the quality of the test sequences:

1. **Strength:**
   - The generated test sequences covered the conformance test suite except for the cases that required hardware interrupts.
   - The auto-test tool identifies four failure cases using 64 test sequences; the conformance test suite for standard status missed three of the four types of test sequences necessary to identify the different causes of system failures.

2. **Weakness:**
   - Checking the functional correctness was not covered by the test automation.
   - Hardware interrupts were not addressed in the test generation.

Tests using the conformance test suite in the standard mode identified one system failure V.1; it failed to identify V.2, V.3, and V.4. In fact, the conformance test suite for extended mode (i.e., debugging mode) included the corresponding test sequences required to identify these missed failures, however, the conformance testing only checks whether illegal call sequences are properly handled by error handling code while it is enabled, and thus, is not as useful to identify undefined behaviors and unsafe codes in the operating system when the debugging mode is disabled. For example, a test sequence that violates CallerMode(ClearEvent, extended) may pass the conformance testing in extended mode by producing an error code, but it causes failure in standard mode since no pre-condition checking exists in the code.

### 7.5. Efficiency of tests

We also evaluated the efficiency of the proposed approach compared to an existing test automation approach, concolic testing (Sen et al., 2005), the state-of-the-art automated test input generation technique that combines symbolic and concrete execution in conjunction with a constraint solver.

Although it is designed for unit testing, we chose concolic testing for comparison because it is fully automated and specialized for fault detection. As concolic testing does not explicitly generate test sequences at the system level, we created a test driver that randomly invoked system calls with random parameters to use concolic testing for system-level testing. In the experiment, we limited the length of the system call sequences to three, for the convenience of analysis, which was sufficient because the four failure cases could be reproduced within three system calls.

We performed concolic testing on Trampoline using CREST (Burnim and Sen, 2008), an open-source concolic testing tool for C. It generated 956 distinct sets of inputs in 9 s and reported 181 input sets that caused failures (Fig. 14). After a manual analysis of executing all 181 cases, 106 cases were identified as false alarms causing no failures. Among the remaining 75, 28 cases were
infeasible sequences that called TerminateTask first and then called other APIs afterwards. After eliminating all infeasible cases and false alarms with a seven-hour manual analysis, 47 cases were finally identified as real failures. However, they were all related to one failure type V.2. In the final result, concolic testing could identify only one out of four failure cases.

In comparison, the test sequences generated from the proposed V&V framework were free from false alarms because it is based on formal models, and were more effective and efficient in identifying real problems.

7.6. Threats to validity

There is a risk that we could miss potential test cases owing to the fixed search depth used in bounded model checking. If comprehensiveness is a key factor, we can increase the search depth until we reach a fixed point of the number of test cases generated, with increasing cost. For example, a test sequence generation for three tasks and three constraints using $CS \times TS_{ad}$ coverage requires 9146, 37235, and 111783 s for search depths 10, 20, and 30, generating 414, 436, and 436 test sequences, respectively. In this case, the search depth 30 is the fixed point because there are no additional test cases generated by increasing the depth. We note that increasing the search depth is expensive; Adding only 5% to the number of test sequences results in a 300% cost increase from depth 10 to depth 20.

Table 3 indicates that the cost of test generation may be exponential to the number of tasks and constraints, especially when we use more rigorous coverage criteria. For example, the time for test generation increases linearly to the number of tasks if we use CS criteria, whereas it increases exponentially if $CS \times TS_{ad}$ criteria is used. This is typical for any comprehensive automated test generation techniques, as the number of test sequences to be generated increases exponentially to the number of tasks and constraints.

Nevertheless, the cost is considered acceptable for an embedded software with less than five to seven tasks, which is a typical case. In exceptional cases with a large number of tasks, yet with stringent resources, we could adjust the cost by choosing less rigorous coverage options, such as CS or $CS \times TS$. We note that the four failure cases were identified using the $CS \times TS$ option.

The test generation was designed for testing the operating system kernel on an ECU and did not attempt to verify multiple ECUs simultaneously. Test generation for control software over multiple ECUs can be accomplished compositionally by reusing the test cases from the unit components (Flemstroem et al., 2015).

8. Related work

The proposed pattern-based V&V framework is founded on various existing ideas including the concepts of specification patterns, configurable model generation, and model-based test generation. This section discusses existing works related to these basic ideas and how they are extended in the proposed framework.

8.1. Pattern-based modeling

Specification patterns can be considered the first patterns used in formal verification (Dwyer et al., 1999; Konrad and Cheng, 2005; Autili et al., 2015). Recently, reference (Filipovikj et al., 2014) studied the usefulness and the potential difficulties in transforming natural language specifications into patterns in the automotive domain. The main aim of specification patterns is to make property specification simple to write. Because it is difficult to write a formal specification of properties in temporal logic from a natural language specification, these approaches identified patterns for commonly used properties and utilized them in model checking. Rather than patternizing properties to be verified, the proposed work extends the idea to patternizing components of the target system and the operational environment of the system.

The origin of using constraint patterns for testing can be found in Carver and Tai (1998) where constraints on event sequencing are used for checking the validity of concurrent programs. Constraint patterns are defined for event sequencing constraints in the propositional model – calculus and validity constraints are derived to check the validity of nondeterministically generated test sequences. Though the idea of using constraint patterns is the same, the patterns are not utilized to generate test sequences. The proposed approach is a domain-specific variation of the original idea and is an advancement to improve the efficiency of test generation.

Reference (Kanstreen, 2009) suggested behavioral pattern mining during test execution to generate models for model-based testing. The proposed V&V framework assumes that commonly agreed requirement specifications are available, such as the OSEK/VDX international standard, and, thus, does not have to mine behavioral patterns from test execution.

8.2. Configurable model generation

Though configurable model generation is a new idea in formal verification, similar ideas have been largely used in product-line engineering using the notion of feature model (Arrieta et al., 2014; Wang et al., 2013). For example, reference (Arrieta et al., 2014) semi-automatically generates plant models in Simulink from feature models that define variabilities and commonalities of the domain of interest. Whereas the product-line approach provides us a general methodology for identifying commonalities and variabilities, and how to utilize them to generate a specific model depending on the choice of variabilities, details on how to apply these techniques on a specific domain are largely left to engineers, which is the most difficult part in practice. The proposed framework provides a practical solution for the verification of small-scale embedded software with predefined behavioral models and parameterized patterns such that the formal model generation is fully automated and highly configurable.

8.3. Model-based test generation

Model-based test generation (Offutt et al., 2003; Wang et al., 2013) has been an active research area owing to its potential for automating test process. Limiting the discussion to domains of embedded systems, references (Fang et al., 2012; Yang et al., 2005; Marinescu et al., 2009) present approaches and case studies on model-based test generation in the domain of automotive embedded systems using various model representations (EAST-ADL, Maude, Promela, etc.), and test generation techniques and tools (UPPAAL, SPIN, NuSMV, etc.). Several works focus on the modeling of OSEK/VDX operating systems, using formal modeling languages such as EAST-ADL (Marinescu et al., 2009), SPIN (Yatake and Aoki, 2010), CSP (Shi et al., 2012; Huang et al., 2011), and Maude
(Zhu et al., 2013), from which model-based test case generation is automated.

Reference (Huang et al., 2011) modeled Tasks and Schedulers and verified properties such as mutual exclusion, excluded priority inversion, and deadlock freedom using CSP. The proposed patterns subsume these models, and, thus, the same properties could be verified in our framework. Reference (Zhu et al., 2013) focused on timing verification such as schedulability, non-fault-propagation, and consistency by modeling only the realtime aspects of the AUTOSAR OS including schedule table and the time frame of interrupt services. The proposed model does not explicitly include the schedule table and the time frame, which are not parts of the OSEK/VDX OS, but is able to represent them by modeling global timers, as a schedule table is simply an extension of alarms facilitating convenient organization of task activation and event setting.

None of above mentioned existing approaches considers methods and tools for automatic construction of formal models.

Property-based testing (PBT) (Arts et al., 2015) is a kind of specification-based testing (Tan et al., 2004) that has been applied for testing automotive software. It randomly generates test inputs from specifications of valid inputs to the system under test (SUT) and requires only property specifications for each module of SUT instead of full functional models. PBT relies on random test generation, and, thus, it is not clear when to terminate testing and does not guarantee to address all reachable states even after millions of test cases. Moreover, PBT generates only valid test cases from the specifications of valid inputs. The proposed pattern-based approach models not only functional behaviors but also constraints of the system, systematically generating test cases for both valid inputs and invalid inputs which are essential for checking safety properties.

8.4. Formal verification

References (Botaschjan et al., 2008; Endres et al., 2010; Klein et al., 2010) provide a rigorous verification framework that proves the correctness and consistency of an operating system at the level of abstract specification and generates code from the specification. The theorem prover HOL/Isabelle is used to check safety issues such as ill-typed pointer access, memory leaks, nontermination, and exceptions, and to prove consistency between different layers of the system, such as consistency between the operating system and the hardware platform or the application software. To apply this approach in practice, however, we must either use a specific operating system that has previously been proven or locate experts in theorem proving to perform the same proof, which can be quite a burden in practice.

9. Conclusion

This paper presented a pattern-based configurable V&S framework for facilitating rigorous V&S activities using formal methods. A major benefit of our framework is that formal models are automatically constructed from a set of predefined patterns, reducing the cost for modeling and eliminating potential human mistakes in modeling process. A prototype tool is developed to demonstrate how suggested framework can be implemented for test generation.

An application of the framework was demonstrated through a series of experiments demonstrating that formal models can be configured and auto-generated, and the test generation using the generated formal models is cost-effective. It cost a maximum of two seconds per trap property and significantly more efficient than the state-of-the-art test generation approach in that it identifies more problems with a considerably less number of test sequences (65 vs. 956). It is also confirmed that the generated test sequences identify more safety issues than the test suites from domain experts.

The patterns presented in this paper are not complete. For example, a pattern for ISRs (Interrupt Service Routines) is not presented mainly because they can be considered as a special type of task. It is also possible that unidentified constraints exist in the OSEK/VDX standard as we manually analyzed the standard. Nevertheless, adding such patterns to the framework is straightforward as it is designed to minimize the couplings among patterns. This is a major benefit of using the proposed framework which allows flexible and continuous evolution.

The framework can complement or add methods and tools used in industry in at least three ways. First, it can complement conformance testing for embedded operating system implementations through the rigorous generation of test sequences, as introduced in this paper. Moreover, it can be used to check the validity of task design for each control program through simulation in conjunction with formal OS models (Park et al., 2016). This task simulation capability is not available in the existing tools such as Simulink or SCADe as they do not consider the underlying operating systems. Finally, it can be a basis for API call safety checking during software development, which is an important issue to ensure the reliability of safety-critical systems (Kim et al., 2016).

Acknowledgment

This work benefited significantly from the prior work performed by Taejoon Byun who built the prototype tool for the test generation. This research was supported by MISP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2016-H850116101200001) supervised by the IITP (Institute for Information & Communications Technology Promotion) and Basic Science Research Program through National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2016R1D1A3B0101685).

References

Yunja Choi is an Associate Professor at Kyungpook National University, South Korea. She received a B.S. degree and a M.S. degree in mathematics from Yonsei University in Korea, and a M.S. and a PhD degree in computer science at the University of Minnesota, USA. She worked as a research scientist at the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany, before joining Kyungpook National University. Her research interests include software safety analysis, light-weight formal methods, and model-driven component engineering.