IEICE TRANS. INF. & SYST., VOL.E100–D, NO.7 JULY 2017

LETTER

A Toolset for Validation and Verification of Automotive Control Software Using Formal Patterns*

Yunja CHOI†a), ***Member and*** **Dongwoo KIM**†, ***Nonmember***

SUMMARY An automotive control system is a typical safety-critical embedded software, which requires extensive verification and validation (V&V) activities. This article introduces a toolset for automated V&V of automotive control system, including a test generator for automotive operating systems, a task simulator for validating task design of control software, and an API-call constraint checker to check emergent properties when composing control software with its underlying operating system. To the best of our knowledge, it is the first integrated toolset that supports V&V activities for both control software and operating systems in the same framework.

key words: *validation, verification, OSEK/VDX, patterns*

1. Introduction

Automotive systems are controlled by numerous Electrical Control Units (ECUs). A controller on each ECU is a result of configuration-dependent compilation of an operating system and application logic implemented by a sequence of collaborating tasks. In such a situation, rigorous V&V activities need to check various aspects to fulfill several goals, including

G1. To validate that a given operating system complies international standards,
G2. To verify that a given operating system is code safe, i.e., does not include software faults which may lead to system failure,
G3. To verify that a given application code complies the constraints of underlying operating system, and
G4. To validate that a given application logic implements the designed task sequence.

There are several tools for supporting V&V activities in this domain [1]–[3]. Most of them considers application programs separately from underlying operating systems, enabling only local reasoning of the V&V activities, or addressing only specific problems, such as schedulability, without giving much thought on checking emergent properties from integrating application software with underlying operating systems. Our previous work identified that a misuse of API functions (provided by operating systems) in application code can be a source of system failure [4].

We present a prototype toolset *AutoCheck*^{FP} to support the overall V&V activities for automotive control systems, aiming at achieving above mentioned goals in one framework. The tool is a result of several years of our collaborative efforts [5]–[7], intended to support (1) auto-generation of formal models for an easy access to formal verification techniques, and (2) an integrated V&V framework for various needs.

The toolset consists of a test generator for checking OS implementations to support the goals G1 and G2, an API-call constraint checker to support the goal G3, and a task simulator to support the goal G4. In the core of the toolset lies a pattern repository, analyzed from the international standard for automotive operating systems, OSEK/VDX [8] and formally modeled as a set of parameterized statemachines [6]. This pattern repository includes both functional behavioral patterns of OSEK/VDX OS and constraint patterns that model prohibited behaviors of application logic defined in the standard. A formal model of an operating system is auto-generated from this pattern repository by instantiating behavioral patterns depending on system configurations, which acts as a core engine to achieve various V&V goals.

The remainder of this paper is organized as follows: Section 2 explains the overall approaches with a motivating example. Section 3 briefly explains the patterns and their usage in the tool. Three major features of the toolset is explained in Sect. 4 followed by a brief discussion in Sect. 5.

2. Motivation and Approach

Figure 1 is a fragment of an application code (left side) together with its configuration (right side). This software consists of two tasks $t1$ and $t2$, one resource $r1$, and two events $e1$ and $e2$. The two tasks defined in the application code interact with underlying operating system through API function calls (lines 03, 05, 09 - 11) and collaborate with each

```
00: // Application code          00: // Configurations
01: int mutex = 0;               01: Task t1 { // no resource
02: Task(t1) {                   02:   PRIORITY = 3;
03:   ActivateTask(t2);          03:   EVENT = e1;
04:   if ( mutex == 0 ) {        04:   EVENT = e2;
05:     WaitEvent(e1); }         05:   AUTOSTART = yes };
06:   ... }                      06: Task t2 { //no event
07: Task(t2) {                   07:   PRIORITY = 1;
08:   if ( mutex == -1 ) {       08:   RESOURCE = r1; };
09:     SetEvent(t1, e1);}       09: RESOURCE r1 {};
10:   ClearEvent(e2);            10: EVENT e1 {};
11:   TerminateTask() }          11: EVENT e2 {};
```

Fig. 1 A code fragment

Manuscript received February 27, 2017.
Manuscript publicized April 19, 2017.
†The authors are with School of Computer Science and Engineering, Kyungpook National University, South Korea.
*The work was supported by the National Research Foundation of Korea Grant funded by the Korean Government (NRF-2016R1D1A3B01011685).
a) E-mail: yuchoi76@knu.ac.kr
DOI: 10.1587/transinf.2017EDL8042

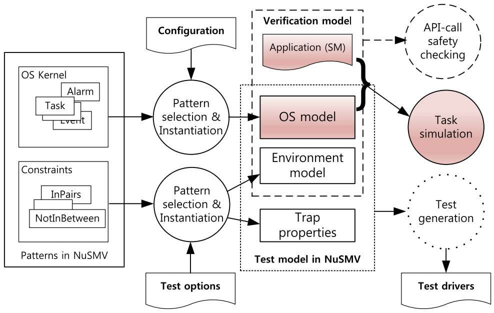

Fig. 2 Overview of *AutoCheck^{FP}*

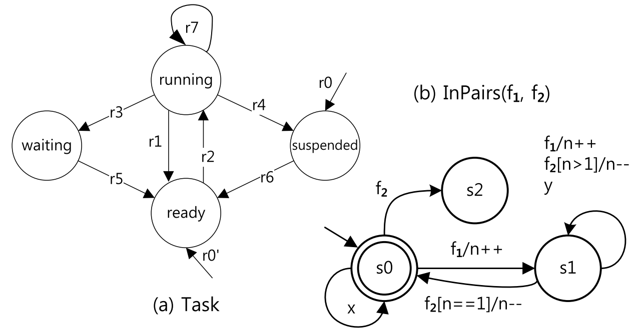

Fig. 3 Sample patterns

other to achieve a required functionality.

An operating system provides major services, such as task management, resource management, event management, and communications, to an application program through API functions. However, interactions between them are not well protected in general so that ill-designed tasks or human mistakes introduced in coding time may lead to unanticipated behavior of the system, including system failure.

For example, line 03 of Task $t1$ activates Task $t2$, but whether $t2$ will preempt $t1$ or not depends on the priority of tasks given by system configurations and the scheduling policy of underlying operating system. Furthermore, the call `WaitEvent(e1)` from Task $t1$ may or may not result in transiting $t1$ into `waiting` state depending on whether $t1$ is an extended task, which owns an event, or not. Checking such behavioral issues and system failures requires considering multiple layers of the ECU software: the application code itself, interfaces between application code and the underlying operating system, and the OS implementation, etc. A software fault from any of these layers can be directly connected to system-level failures.

Motivated by failure cases identified from our previous case studies [4], [6], we have developed a prototype toolset *AutoCheck^{FP}* based on a pattern-based model generation framework [9]. Figure 2 is an overview of the toolset. The base part of the toolset is a pattern repository, pre-defined for modeling OSEK/VDX operating system kernel and for operational constraints on application programs identified from the OSEK/VDX international standard and formalized in the input language of the model checker NuSMV. These OS patterns and constraint patterns are composed depending on system configurations to generate a test model or verification model. The toolset performs model-based test generation, task simulation, or code-level API-call safety checking from these models using model checker NuSMV and CBMC as backend V&V engines. *AutoCheck^{FP}* provides configurable multi-layer verification framework, from operating systems to control software, fully utilizing formal models and formal verification engines.

3. Pattern-Based Model Construction

This section briefly summarizes two representative patterns and their models in NuSMV to help readers to understand the formal model construction process.

3.1 Patterns for OSEK/VDX OS

Figure 3 (a) illustrates a statemachine representation of a task with four states and eight transitions. Each transition is triggered by an API function call from application software under specific conditions and may perform a set of actions as the result of transition. Specific conditions for a transition may be determined by states of other OS constructs and system configuration which become parameters of the pattern. Below is the declaration part of the Task pattern in NuSMV, illustrating how it is parameterized:

```
MODULE Task(env, tid, ptiv, pri, autostart,
             extended, rq, e_run, res, evt)
  VAR  state : { SUS, RDY, WIT, RUN};
```

In the declaration, *tid*, *ptiv*, *prio*, *autostart*, *extended* are parameters from system configuration and *env*, *rq*, *res*, *evt* are parameters from other statemachines, representing Application, Scheduler, Resource, and Event, respectively. The `VAR` statement declares variables in the statemachine, representing the four states of a Task in this case. Each transition of Task pattern is specified with `ASSIGN` statements in the `MODULE` as follows:

```
// initialize the state variable
init(state) := case autostart : RDY;
                    TRUE      : SUS;
                esac;
// transition r2
next(state):=  case
  (state = RDY & !e_run & prio >=rq.max_prio ) ||
  (state = RDY & (env.nSC | env.nRR | env.nSE)  &
  rq.pq[prio][0]= 1 & prio >= rq.max_prio : RUN;
     ...
esac;
```

API function calls are encoded with abbreviations, e.g., nSC, nRR, and nSE, representing `Schedule`, `ReleaseResource`, and `GetResource`, respectively. The r_2 transition specifies that there are two cases a task transits from `Ready` state to `Running` state: (1) if the task is in `Ready` state, no other task is in running currently, and the priority of the task is greater than or equal to the maximum priority of tasks in the ready queue, or (2) if the task is in `Ready` state, other task calls one of `Schedule`, `ReleaseResource`, `SetEvent`, and there is a task in the queue head whose priority is the largest among all the tasks in the ready queue.

Our pattern repository includes patterns for basic constructs of OSEK/VDX OS, such as Tasks, Schedulers, Resources, Events, and Alarms, formalized as parameterized statemachines, $M^T[C_T]$, $M^S[C_S]$, $M^R[C_R]$, $M^E[C_E]$, and $M^A[C_A]$, where C_x denotes a set of configuration-dependent parameters for each statemachine. Given system configuration $C = C_T \cup C_S \cup C_R \cup C_E \cup C_A$, a formal OS model is generated as a synchronous parallel composition of a set of parameterized statemachines:

$$OS[C] = M^T[C_T] || M^S[C_S] || M^R[C_R] || M^E[C_E] || M^A[C_A]$$

The number of each type of statemachine to be composed is also dependent on the system configuration. For example, if two tasks are declared, two instances of M^T are to be composed in the OS model.

3.2 Constraint Patterns

AutoCheck^{FP} innovatively models operational constraints identified from the OSEK/VDX standard in formal patterns. The purpose is to formally specify prohibited behaviors of application software and use the patterns to rigorously verify the interactions between any application software and its underlying operating system. Table 1 is a selected list of constraint patterns among a total of 13 constraint patterns we have defined. For example, the `InPairs` constraint pattern abstracts constraints that impose pairwise calls to API functions, such as InPairs(GetResource, ReleaseResource). A representative example of the `OwnerOnly` constraint is `WaitEvent(e)`, where the caller of the function must own the event *e*.

Figure 3 (b) illustrates a formal representation of `InPairs(`f_1`, `f_2`)` constraint; s_0 is the initial and the final states, where f_1 and f_2 are paired, s_1 represents a state where the number of calls to f_1 is greater than that of calls to f_2, and s_2 is a state where the number of calls to f_2 exceeds that of calls to f_1 which is an error state.

Table 1 Sample constraint patterns

Patterns	Constraint Description	category
InPairs(f_1, f_2)	f_1 and f_2 shall be called in pairs in the order of f_1 followed by f_2.	call seq.
NotInBetween(A, f_1, f_2)	A call in a set A shall not be called in between calls to f_1 and f_2.	call seq.
MustEndWith(A)	A call in a set A shall be called eventually and no calls shall be followed afterwards.	call seq.
CallerMode(f, m)	If the API call is f, the the mode of the caller task shall be m.	config.
OwnerOnly(f)	The caller task of f should own the object referenced by f.	config.

4. Features of the Toolset

4.1 Task Simulator

It is important to be able to validate that the application code truthfully implements the expected task execution sequences designed for performing a specific functionality. Currently available commercial tools do not provide means to validate such task sequences at the software level as they do not take underlying operating systems into account.

The task simulator in *AutoCheck^{FP}* supports early validation of task sequences before the code is compiled with operating system. The application code is abstracted w.r.t. the API function call sequence and is translated into a NuSMV module M_{app} which is then composed with the configuration-dependent OS model generated from the tool to construct a simulation model: $M_{sim}[C] = OS[C] || M_{app}$.

The toolset uses the model checker NuSMV as the simulation engine and visualizes task sequences. Figure 4 shows the visualization of a task sequence of the code fragment in Fig. 1.

4.2 Test Generator

To achieve the goals G1 and G2, the toolset includes a pattern-based test generator. The test generator constructs a test model for given system configuration and test option which is a selection of constraint patterns. A test model is a synchronous parallel composition of an OS model and a set of selected constraint patterns.

$$M_t[C] = OS[C] || M_{p_1} || M_{p_2} || \dots || M_{p_n}$$

Once the test model is constructed, the tool uses the typical model-based test generation strategy: It generates a set of trap properties stating that each state of the test model is not reachable. For a test model with a task and *InPairs* constraint as shown in Fig. 3, for example, various trap properties can be set;

TR1. The *InPairs* pattern never reaches to the state s_1 (Constraint Cover), or

TR2. The task and constraint statemachines can never be *running* and s_1 states at the same time (Task & Constraint Cover).

These trap properties are generated in temporal logic LTL

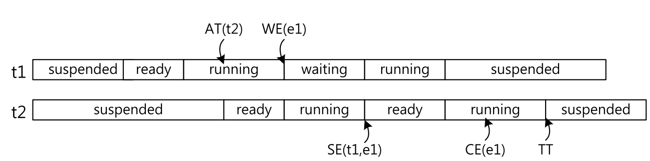

Fig. 4 A visualized task simulation

Table 2 Performance of test generation

	Constraint Cover		Constraint & Task Cover	
(T, C)	Time	#P(#S)	Time	#P(#S)
(2, 2)	4.63	64(24)	16.24	128(45)
(2, 3)	39.46	88 (31)	28.32	176(53)
(2, 4)	40.48	104(34)	32.15	208(58)
(3, 2)	21.24	132(50)	226.24	704(218)
(3, 3)	56.28	144(59)	370.71	896(253)
(3, 4)	33.63	192(65)	412.69	1024(265)
(4, 2)	64.42	224(92)	6289.91	3584(817)
(4, 3)	103.97	272(109)	3766.22	4352(1102)
(4, 4)	105.31	304(116)	4266.78	4864(1152)

and are either verified (if they are not reachable indeed) or refuted with counterexamples using NuSMV as a backend verification engine. The tool converts counterexamples into test drivers for testing actual OS implementations.

Table 2 shows the performance of test generation in seconds as the numbers of tasks and constraints increase from 2 to 4. (T, C) represents the combination of numbers of tasks and constraints, #P and #S represent the number of trap properties generated and the number of final test sequences, respectively.

Test efficiency of the tool is compared with the state-of-art test generator using concolic testing on an open source operating system Trampoline; Testing using the test cases generated from $AutoCheck^{FP}$ could identify more failure types (four vs. one) and does not include infeasible test sequences or false alarms. On the other hand, concolic testing using the tool CREST generates many infeasible test sequences (28.3%) and false alarms (11%), identifying only one type of failure cases [4].

4.3 API-Call Constraint Checker

The API-call constraint checker is to ensure the correctness of the interaction behavior between the OS and application software as stated in the goal G3. For a given application code, the checker verifies whether the code complies operational constraints of OSEK/VDX OS w.r.t. the 13 constraint patterns in the pattern repository.

The tool provides two options, one for checking local constraints, which applies within a task, and the other for checking global constraints, which involves multiple tasks. For checking local constraints, $AutoCheck^{FP}$ annotates each task with calls to monitoring code which is a pre-declared constraint automaton as a C-library function. The C code model checker CBMC is then used to check whether the task terminates while the constraint automaton is in unsafe (non-terminal) state. For checking global constraints, the toolset extracts statemachine representation of the application code and constructs a synchronous parallel composition of a configuration-dependent OS model, the statemachine representation of application code, and a constraint pattern: $M_{API}[C] = OS[C] || M_{app} || M_p$. NuSMV is used to check whether unsafe state of the constraint model can be reached in M_{API}.

Table 3 shows the performance of constraint checking

Table 3 Performance of API-call constraint checking

Constraint ID	1	2	3	4	5
Global Checker	39.61	11.90	37.72	11.80	15.01
Local Checker	0.011	0.011	0.008	0.017	N/A

when 15 application programs are checked against 5 different local/global constraints [7]. Local constraint checking is much faster, but misses violations of global constraint 5.

5. Conclusion

We have presented a prototype toolset $AutoCheck^{FP}$ implemented using formal pattern repository. The toolset has been developed to demonstrate that formal approaches can be practical and beneficial in multiple ways. Major benefits of our approach include that (1) configurable formal models can be auto-generated, (2) formal patterns can be reused in other similar domains, (3) it supports multi-layer V&V activities, including task simulation, test generation, and formal verification, within an integrated framework, (4) it increases the accuracy of V&V by taking underlying OS behavior into account, and (5) the pattern-based nature of the tool makes it flexible and extensible to support other V&V activities than those presented in this work.

Though the toolset is not in public space, demonstrations are available at [10], [11].

References

[1] C. O'Halloran, "Automated verification of code automatically generated from Simulink," Automated Software Engineering, vol.20, no.2, pp.237–264, June 2013.
[2] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing AUTOSAR software with QuickCheck," IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, 2015.
[3] E. Technologies, "Scade suite," http://www.esterel-technologies.com.
[4] T. Byun and Y. Choi, "Automated system-level safety testing using constraint patterns for automotive operating systems," Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, pp.1815–1822, 2015.
[5] Y. Choi, M. Park, T. Byun, and D. Kim, "Efficient safety checking for automotive operating systems using property-based slicing and constraint-based environment generation," Science of Computer Programming, vol.103, pp.51–70, 2015.
[6] Y. Choi and T. Byun, "Constraint-based test generation for automotive operating systems," Software and Systems Modeling, vol.16, no.1, pp.7–24, Feb. 2017.
[7] D. Kim, Y. Chung, and Y. Choi, "Model-based API-call constraint checking for automotive control software," Asia-Pacific Software Engineering Conference, 2016.
[8] OSEK/VDX operating system specification 2.2.3.
[9] Y. Choi, "A configurable V&V framework using formal behavioral patterns for automotive control software," Journal of Systems and Software, submitted.
[10] SSELAB, AutoCheck^{FP}, Available at http://sselab.dothome.co.kr/wordpress/index.php/tool-demo-autocheck-fp/.
[11] SSELAB, "A tool demonstration of NuSek test generation," Available at http://sselab.dothome.co.kr/wordpress/index.php/tool-demo-nusek.