Property-based Code Slicing for Efficient Verification of OSEK/VDX Operating Systems*

Mingyu Park   Taejoon Byun   Yunja Choi

School of Computer Science and Engineering
Kyungpook National University
Deagu, Korea

pqrk8805@gmail.com   bntejn@gmail.com   yuchoi76@knu.ac.kr†

Testing is a de-facto verification technique in industry, but insufficient for identifying subtle issues due to its optimistic incompleteness. On the other hand, model checking is a powerful technique that supports comprehensiveness, and is thus suitable for the verification of safety-critical systems. However, it generally requires more knowledge and cost more than testing. This work attempts to take advantage of both techniques to achieve integrated and efficient verification of OSEK/VDX-based automotive operating systems. We propose property-based environment generation and model extraction techniques using static code analysis, which can be applied to both model checking and testing. The technique is automated and applied to an OSEK/VDX-based automotive operating system, Trampoline. Comparative experiments using random testing and model checking for the verification of assertions in the Trampoline kernel code show how our environment generation and abstraction approach can be utilized for efficient fault-detection.

1   Introduction

The operating system is the core part of automotive control software; any malfunction can cause critical errors in the automotive system, which in turn may result in loss of lives and assets. Testing has been widely used as a systematic and cost-effective safety analysis/assurance method [6, 18], but its optimistic incompleteness often misses critical problems and cannot guarantee the "absence of wrong behavior". As an alternative and complimentary technique, model checking [10, 16] has been drawing attention from both academia and industry.

Model checking is a comprehensive formal verification technique, suitable for functional safety analysis. It can effectively identify subtle issues, such as process dead lock, illegal behavior, and starvation, but may require more resources and domain knowledge. In particular, the use of model checking faces the following challenges:

1. The size of model/code to be verified needs to be minimized to avoid state-space explosion.

2. Modeling of the environment, such as user tasks and hardware environment, is necessary and critical for embedded software.

Since an operating system is a reactive system responding to environmental stimuli, the correctness of its behavior needs to be analyzed with respect to the behavior of its environments. A non-deterministic environment is typically used to over-approximate actual behavior, but it is often too expensive in model

---

checking. The difficulty and importance of defining a *good* environment model has been addressed in a number of previous works [22, 23, 12, 25, 17, 20].

We note that these two problems apply to both model checking and testing. Though the level of comprehensiveness differs, both techniques rely on automated search techniques that are initiated by environmental stimuli. This is called environment model in model checking and test scenario in testing. This work anticipates that the efficiency of automated verification techniques depends on the modeling of the environment and proposes an application of property-based code slicing [24] for automatically generating an environment model using the data/function dependency analyzed from the operating system kernels. The goal is to construct a valid and comprehensive usage model of the operating system with minimal dependency on the kernel code.

Our approach extracts functions that have a direct dependency on a given property to be verified and generates non-deterministic function-call sequences by imposing (1) external constraints from the OSEK/VDX standard [1] for automotive operating systems, and (2) internal constraints identified from the function call structure of the operating system kernel. The external constraints are manually identified from the specifications of the standard and are imposed on the initially random sequence of function calls. The internal constraints are imposed by identifying the top-level functions using backward slicing from a given property and by computing the cone-of-influence from each top-level function using forward slicing. The Environment model is defined as an arbitrary sequence of calls of those extracted functions. The operating system kernel is also abstracted as a collection of extracted functions and its relevant code required for pre-processing them. This procedure reduces the size of the verification target and minimizes the behavior of the environment model. The extraction and model construction process is automated with the aid of the static analysis tool Understand [4].

The approach and the tool are applied to the verification of safety properties of the Trampoline operating system [3], which is an open source automotive operating system compliant with OSEK/VDX. Environment models are generated using the assertions identified from the kernel code, and the kernel code itself is reduced by including only those extracted functions and their relevant code. The environment model is used to model-check/test the abstract code using CBMC [9] and random testing. We compare their fault-detection capability, their comprehensiveness in terms of code coverage, and their efficiency in terms of resource consumption.

The remainder of this paper is organized as follows. Section 2 briefly discusses related work and Section 3 provides the motivation for our work. Section 4 provides an overview of our approach and Section 5 presents the methods and the process for the automated environment generation technique. Section 6 explains the environment settings for the collaborative verification, followed by experimental results and the evaluation using Trampoline OS as a case example in Section 7. We conclude in Section 8.

2   Related Work

Environment modeling for efficient model checking has been an active research issue [22, 23, 12, 25, 17, 20]. Reference [20] is one of the earliest works concerning environment assumptions in verification. It introduced the *observer* concept to represent assumptions about the environment. The approaches for assumption generation were developed further in [12, 23, 13, 19]. Reference [23] automatically generates the environment of Java programs from the specifications written by a user. [19, 13] are concerned about automatic partitioning, learning, or minimizing assumptions for compositional verification. None of them considers environment generation for both model checking and testing.

Several specification-based environment generation methods exist: [21] uses ADL to define pro-

tocols of Java components and constructs an environment for the ADL specification. [11] describes environmental assumptions in LTL and uses them to filter a universal environment, which is adopted in our approach to constrain the non-deterministic initial task model. Reference [25] automatically generates scripts in PROMELA from environment models for OSEK/VDX-based operating systems that are modeled in UML diagrams. Their approach, however, models all basic objects in OSEK/VDX using UML class diagrams and state diagrams, from which all combinations of deterministic environments are generated and verified individually. The models are then used to automatically generate exhaustive test cases for the conformance testing of OSEK/VDX-compliant operating systems [8]. Their approach assures the exhaustiveness of test cases, but the scalability issue remains, as the number of test cases may increase exponentially.

Program slicing [24] has been a popular technique for reducing verification complexity for both model checking and testing. References [5] and [14] use slicing algorithms to explicitly detect $def-use$ associations that are affected by a program change for efficient regression testing. Reference [7] performs program slicing for C programs with respect to the alarms generated from value analysis. [15] integrates aggressive program slicing and a proof-based abstraction-refinement strategy for wireless cognitive radio systems. It is a representative example of using program slicing and bounded model checking for embedded software, but the slicing is integrated into the model checking process, and is thus not suitable for application in testing.

3 Background

3.1 OSEK/VDX

OSEK/VDX is a joint project of the automotive industry, which aims at establishing an industry standard for an open-ended architecture for distributed control units in vehicles [1]. The aim of OSEK/VDX is to provide standard interfaces independent of application, hardware, and network, and ultimately, to save the development costs for non-application related aspects of control software. It is specialized for automotive control systems, removing all undesired complexities such as dynamic memory allocation, circular waiting for resources, multi-threading, and so on. Since its target system is safety-critical, it strictly prohibits uncontrolled dynamic behavior of the system.

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

3.2 Trampoline

Trampoline [3] is an open source, real-time operating system compliant with OSEK/VDX version 2.2.3. It is developed in ANSI C and can be ported to various hardware platforms such as Arm, POSIX, PPC, AVR, HCS12, C166, etc. Since it also supports POSIX, it can be test-run on a UNIX/Linux environment before being ported to an actual operational environment. As its target platform varies, its platform-dependent part is clearly structured in a separate module that combines with the kernel module at compile time. Access to the hardware-specific part is abstracted using *extern* variables and macros so that the main control logic does not need to be aware of the specific hardware feature. As illustrated in
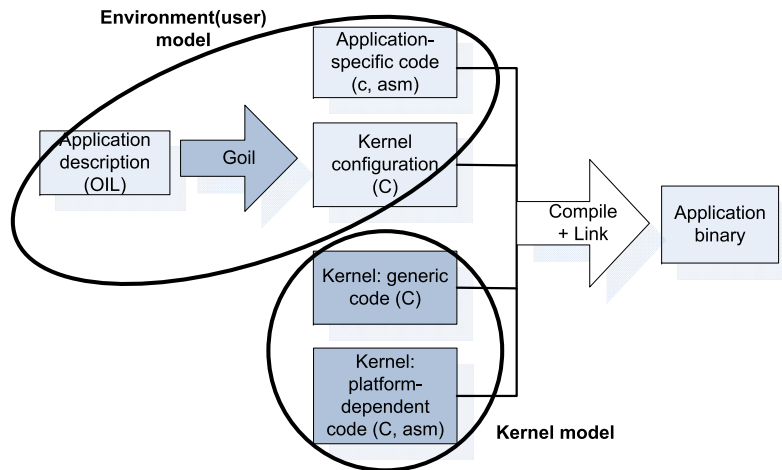
Figure 1: Components of Trampoline

Figure 1, development of an automotive software using Trampoline requires four components; (1) application source code, (2) kernel configuration generated from configuration description written in OIL (OSEK Implementation Language) using the Goil compiler, (3) generic OS kernel code compliant with the OSEK/VDX standard, and (4) platform-dependent kernel code. The generic OS kernel code implements services for task management, resource management, interrupt handling, and event/counter/alarm management, providing corresponding APIs.

3.3   Model checking using CBMC

Formal verification methods based on model checking [10] are an effective technique for identifying subtle issues in software safety which is particulary important for embedded systems. Current technological advances in model checking enable engineers to directly apply the technique to program source code, removing the manual model construction process. CBMC [9] is one of these model checking tools, which is capable of verifying almost full ANSI C. It can be used to verify buffer overflows, pointer safety, exceptions and user-specified assertions. Furthermore, it can check ANSI C and C++ for consistency with other languages, such as Verilog. The main advantage is that it is completely automated and generates counterexample traces when a property in question is refuted.

    As with any other model checking tool, CBMC also suffers from the problem of scalability. When applied to the Trampoline kernel as a whole with an arbitrary sequence of API calls, for example, it ran out of memory for checking one assertion on a PC with 3GB of memory.

4   Overall Approach

Comprehensive verification, required by functional safety analysis, is too costly to be applied in practice. Reducing the cost while maintaining comprehensiveness is a challenging, but crucial task. Our approach attempts to achieve this goal with the following three strategies:

1. Property-based environment generation: An environment of the operating system kernel is automatically generated using static code analysis for a given safety property.
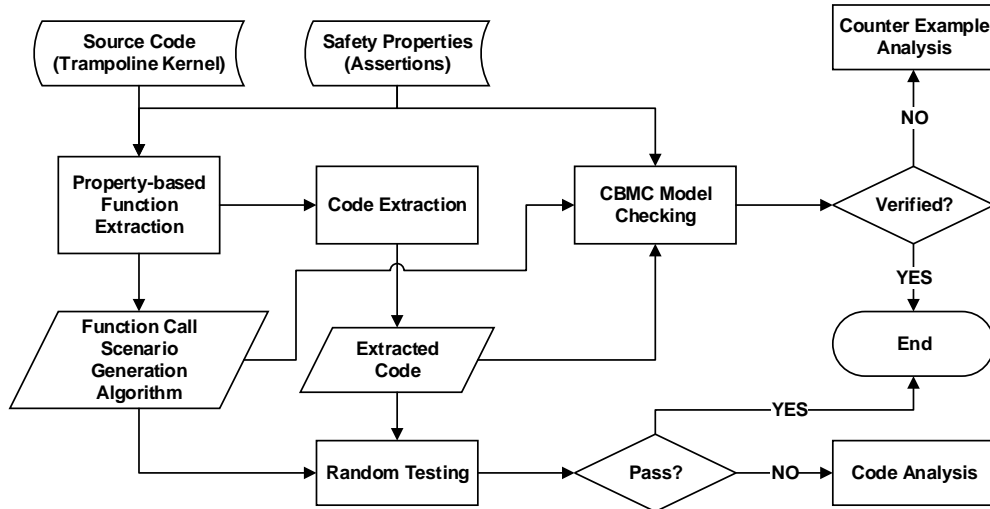
Figure 2: Collaborative verification approach

2. Property-based abstraction: The operating system kernel is abstracted by extracting only the code relevant to a given property.

3. Collaborative verification using model checking and testing: Both model checking and testing are used complementarily for the verification of the abstract kernel code under the generated environment model.

Cost reduction is achieved through property-based environment generation and code abstraction. The efficiency of verification is increased by taking advantage of both verification techniques. Figure 2 is an overview of the suggested collaborative verification approach. Our approach uses both model checking and testing to complimentarily utilize their different capabilities when only limited resources are available.

5   Environment Generation

A straightforward way to include all possible task interactions with the operating system is to model the task with strongly connected states, where each state represents an API call to the kernel and each transition between states is not guarded. However, this includes too many spurious and/or impossible behaviors and increases the cost for verification as well as counterexample analysis; if 26 APIs are provided by the operating system, the task model would have at least 26 strongly connected states. Our approach tries to minimize unnecessary verification cost by using property-based extraction of dependent functions.

5.1   Abstraction through static code analysis

Given a property, we first extract the variables specified in the property, which is called *Verification Target Variables*, and identify all the variables that are used to define the *Verification Target Variable*, called *Extended Verification Target Variable*. Then, functions modifying those *Extended Verification Target Variables*, called *End_Level_Functions*, are extracted. The prototypes of the *End_Level_Functions* are
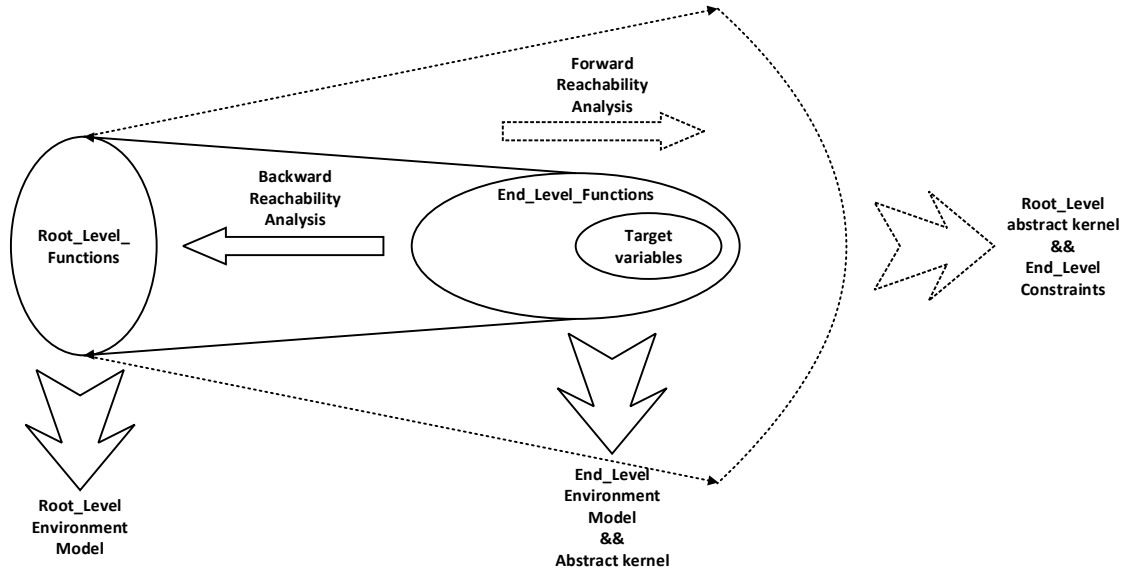
Figure 3: Backward and forward reachability analysis for environment generation

used to construct an end-level environment model. The corresponding end-level abstract kernel code consists of all the *End_Level_Functions* and their dependent code. The *Root_Level_Functions* are identified by performing backward reachability analysis from each *End_Level_Function*. The prototypes of the *Root_Level_Functions* are used to construct a root-level environment model. Its corresponding abstract kernel code is identified by performing forward reachability analysis from each *Root_Level_Function*. The result of forward reachability analysis is also used to identify constraints for the end-level environment model.

Definition 1 *Property-related variables:*

1. *A Verification Target Variable is a variable that appears in the property specification.*

2. *An Extended Verification Target Variable is a variable that a Verification Target Variable depends on.*

Definition 2 *Classification of functions:*

1. *An End_Level_Function is a function that directly modifies, sets, or uses an Extended Verification Target Variable.*

2. *A Root_Level_Function is an API that is a terminal node of the called-by graph of an End_Level_Function.*

From this process, we extract two types of functions for constructing different levels of environment models: (1) functions for root-level environments, and (2) functions for end-level environments. Figure 3 shows the conceptual diagram for the whole process.

For a simple example, if a property in question is

$$Property_1 : \ assert(tpl\_fifo\_rw[tpl\_h\_prio].size > 0),$$

then we first identify *Extended Verification Target Variables* and *End_Level_Functions* for *tpl_fifo_rw* and *tpl_h_prio*. The identified set of *End_Level_Functions* for the variable *tpl_h_prio* is { *tpl_get_proc*, *tpl_put_preempted_proc*, *tpl_put_new_proc*, *tpl_schedule_from_running* } in the Trampoline kernel. An end-level environment model is constructed as non-deterministic calls to those end-level functions and its corresponding abstract kernel encompasses all the identified *End_Level_Functions* and their dependent code. We then identify *Root_level_Functions* for each of the *End_Level_Functions*; For examples, {*ReleaseResource*, *Schedule*, *ActivateTask*, *SetEvent*, *TerminateTask*, *ChainTask*, *WaitEvent*, *StartOS*} are *Root_Level_Functions* for *tpl_get_proc*, which are identified from its called-by graph. A root-level environment model consists of non-deterministic calls to those API functions and its corresponding abstract kernel encompasses all the identified root level functions and their dependent code.
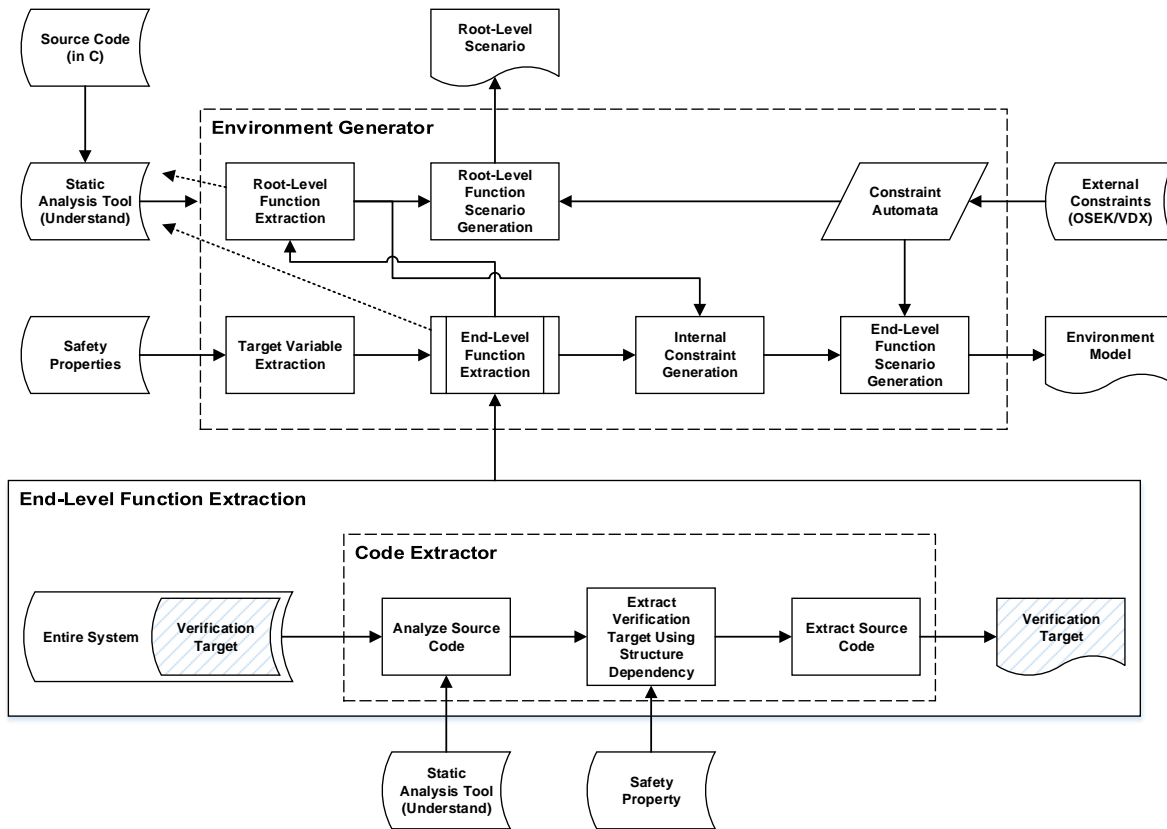
5.2   Implementation

Figure 4: Environment Generation

The suggested approach was implemented and fully automated. Figure 4 shows the overall structure of the automation. The source code of the Trampoline OS is analyzed by the static analysis tool Understand [4], which creates a data repository of the analysis results from which information on variable/function dependencies can be extracted using a C plug-in. The environment generator first extracts target variables from the properties and then extracts Extended Target Variables and End-Level Functions by analyzing dependency relations among variables and functions. Root-Level Functions are then

extracted from the called-by graph for each End-Level Function. A Root-Level Function Scenario is generated as an arbitrary sequence of function calls of Root-Level Functions, which complies with the external constraints from the OSEK/VDX standard. Finally, the environment model is generated as an arbitrary sequence of End-Level Functions that complies with both the external and the internal constraints. The internal constraints are partial-order relations among End-Level Functions, which are generated from each Root-Level Function.

The last step is the property-based abstraction of the original code. Since the environment generation step identifies all necessary End-Level Functions modifying the Verification Target Variables together with the ordering relation among them in call sequences, verification requires only the source code of those End-Level Functions plus codes for preprocessing them. Therefore, for each safety property, verification is performed using the environment model generated with the tool and the property-based abstract code.

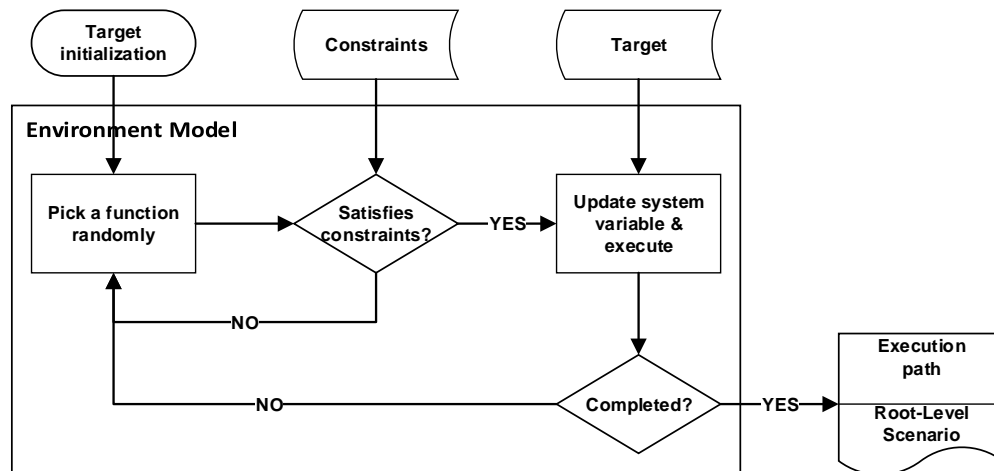## 6   Setting up Environments for Collaborative Verification

Figure 5: Scenario generation process for random testing

CBMC and testing require different settings for their verification environments even with the same set of *End_Level_Functions*; CBMC requires only the algorithm of non-deterministic function calls. Random testing (both root-level and end-level), however, requires explicit function call sequences generated from a given environment model. Figure 5 shows the process for generating such function call sequences, which can be applied to both End-Level and Root-Level environment models. It repeats the selection and checking process by arbitrarily selecting a function and checking whether the selected function satisfied the constraints or not. We have implemented an OSEK/VDX simulator to check the external constraints in the process of root-level sequence generation. Internal constraints are identified by call graph analysis. Details are described in the following two sub-sections.

We note that the checking the constraints is not necessary to ensure the correctness of the scenario generation, but it is essential for making the verification efficient, since otherwise the verification produces too many false errors.
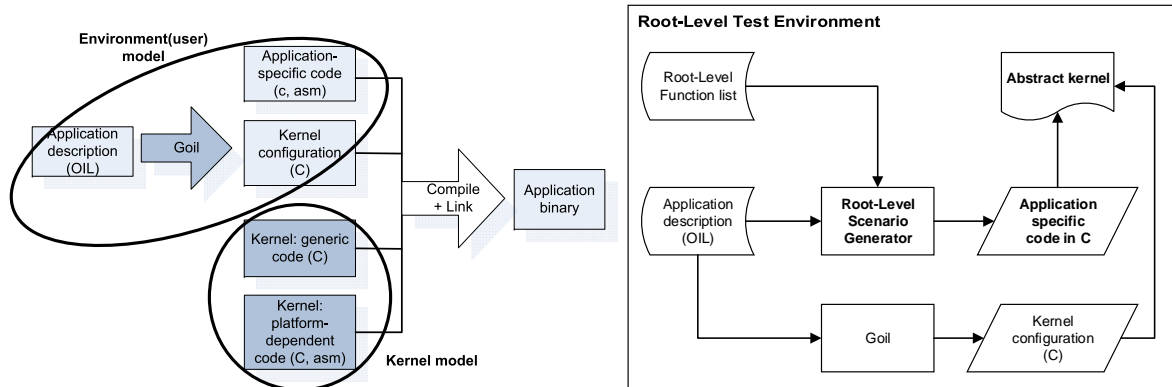
6.1 Root-Level Scenario Generation

Figure 6: Root-Level scenario and the testing environment

Root-Level scenario generation is based on the scenario generation process illustrated in Figure 5. A Root-Level Function Scenario is an arbitrary sequence of function calls of Root-Level Functions, and it complies with the external constraints from the OSEK/VDX standard. Figure 6 shows the Root-Level test environment for the Trampoline OS. Since an OIL file is required for testing, an OIL file is specified as an input, which is compiled with the extracted kernel code and the random sequence of Root-Level Functions generated from our Root-Level scenario generator.

There are two things to be done before implementing a scenario generator. First, every constraint specified in the OSEK/VDX standard should be identified. Second, an OSEK/VDX Simulator - an abstract OSEK/VDX model - should be implemented in order to trace all the changes and fully observe constraints.

6.1.1 Identification of external constraints

The OSEK/VDX standard explicitly specifies constraints among the APIs. The *description* column of Figure 7 lists some of the constraints manually identified from the standard. These constraints are represented as pre-conditions with respect to other APIs. For example, the API function *TerminateTask* can be called only if the task has been activated either by *ActivateTask* or *ChainTask*. Therefore, we set {*ActivateTask*, *ChainTask*} as preconditions of *TerminateTask*. Figure 7 shows a couple of preconditions of other API functions.

Identified constraints are then imposed in the Root-Level scenario generation process illustrated in Figure 5. Figure 8 is an example algorithm of the constraint checking.

6.1.2 OSEK/VDX Simulator

To fully consider all the identified constraints, it is necessary to trace changes that previous function calls have made. For example, if *ActivateTask(t1)* is chosen as the first Root-Level Function in a scenario, task *t1* should be marked as *READY* task, for further scenario validation. This process is fully automated by implementing an OSEK/VDX simulator.

The OSEK/VDX simulator traces run-time information such as list of resources, list of events, list of task models, reference to running task, ready queue (priority queue), and waiting queue. It provides Root-

API function	Pre-condition	Description
StartOS	!StartOS	*StartOS* is executed only once at the beginning.
WaitEvent	!WaitEvent || SetEvent	Since the state of calling task is set to *WAITING* after calling *WaitEvent*, it cannot take any action until *SetEvent* is called. So *WaitEvent* can only be called when it has first appeared, or every waiting task has been set by *SetEvent*.
ReleaseResource	GetResource	*ReleaseResource* can be called when the resource is already held.
TerminateTask	ActivateTask || ChainTask	*TerminateTask* can be called only when there is running task, which is activated by *ActivateTask* or *ChainTask*.
Schedule	!GetResource	*Schedule* can be called only when there is held resources, which is held by *GetResource*
ChainTask	!GetResource	*ChainTask* can be called only when there is held resources, which is held by *GetResource*

Figure 7: Constraint list extracted from OSEK/VDX spec

```
bool canReleaseResource( Resource r1 ){
        if ( running task doesn't exist )
                return false;
        else if ( running task is not holding resource r1 )
                return false;
        else    // when running task is holding the resource r1
                return true;
}
```

Figure 8: ReleaseResource constraint checker

Level Function calls just like OSEK/VDX APIs. When one of these procedures is called, it simulates the behavior of OSEK/VDX. The simulator includes task model, task scheduler, event management, and resource management. Figure 9 shows the overall process of the OSEK/VDX simulator.

A randomly chosen function is added to the body of the running task by a module called Scenario Generator. When there is a function in the body of the running task that has not been executed by the simulator, it executes the function. If scheduling is necessary, it calls the scheduler and executes all the functions that already exist in the body of the preempted task, when preemption occurs. This process is repeated until there is no function left to execute, and the Scenario Generator is requested to generate another function randomly.

Whether scenario generation has been completed or not can be determined by checking all tasks that are initially generated from the given OIL file. If every task has completed its execution with *TerminateTask* or *ChainTask*, it can be determined that the scenario generation is completed.

6.2 End-Level Environment Model

The End-Level scenario generation process is also based on the scenario generation process illustrated in Figure 5. It executes an arbitrary sequence of *End_Level_Functions*, which is chosen from among the list of *End_Level_Functions* serving as an environment of CBMC model checking and End-Level random testing.

This process considers internal constraints among End-Level functions; since End-Level functions
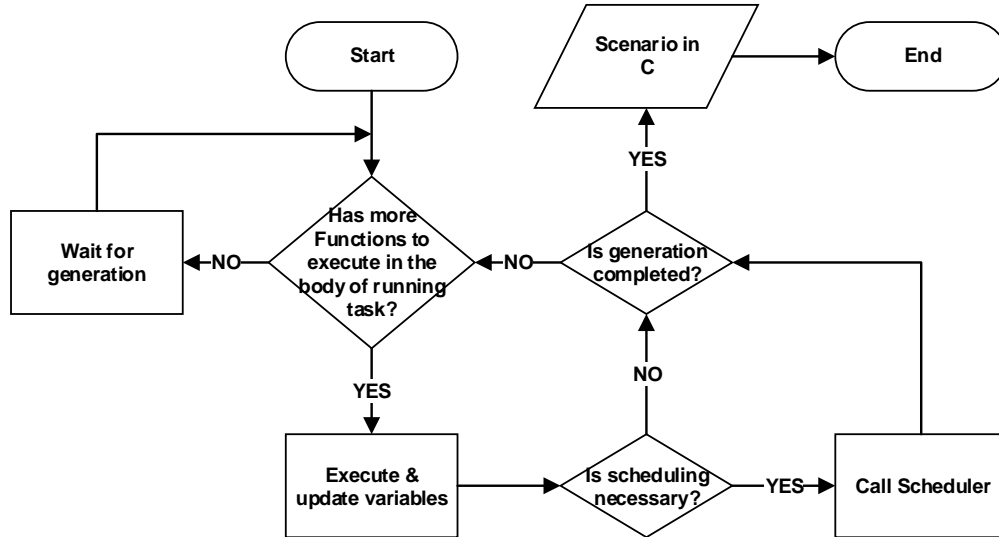
Figure 9: OSEK/VDX simulator

can only be called by API-Level functions, the pre-conditional constraints of Root-Level functions identified from the OSEK/VDX constraints must be implicitly obeyed by the End-Level functions. These implicit constraints can be identified by analyzing call-graphs of Root-Level functions and their pre-conditional relations. We call such implicit constraints internal constraints.

API	End-Level Function
StartOS	tpl_put_new_proc, tpl_put_new_proc, tpl_get_proc
WaitEvent	tpl_get_proc
SetEvent	tpl_put_new_proc, tpl_schedule_from_running
ReleaseResource	tpl_schedule_from_running
ActivateTask	tpl_put_new_proc, tpl_schedule_from_running
TerminateTask	tpl_get_proc
Schedule	tpl_schedule_from_running
ChainTask	tpl_put_new_proc, tpl_get_proc

Figure 10: End-Level Functions called by each API-Level Function

For example, if we consider the Root-Level APIs and their corresponding End-Level functions, the external constraint *WaitEvent can be called after SetEvent is called* can be re-interpreted as tpl_get_proc *can be called after* $tpl\_put\_new\_proc$ *and* $tpl\_schedule\_from\_running$ *are called*, which can be written in a regular expression $(tpl\_put\_new\_proc\ tpl\_schedule\_from\_running\ (End\_Level\_Function - tpl\_get\_proc)^*\ tpl\_get\_proc)^*$.

Checking constraints of this kind cannot be done using the OSEK/VDX simulator because the End-Level functions include implementation-specific function names that cannot be modeled from the standard. Instead, the internal constraints are simplified using the characteristic that *a function cannot be called more times than its preceding functions in the partial order relation*. The example internal constraint can be simplified as *The number of* tpl_get_proc *calls cannot exceed the one of either*

$tpl\_put\_new\_proc$ or $tpl\_schedule\_from\_running$. The constraint checker keeps track of the number of each End-Level functions calls and checks ($\#tpl\_get\_proc < \#tpl\_put\_new\_proc$) && ($\#tpl\_get\_proc < \#tpl\_schedule\_from\_running$).

7   Experiments

We have conducted a series of experiments to show the impact of our approach using CBMC model checking, End-Level random testing, and Root-Level random testing. The target verification properties are three functional safety properties from the Trampoline kernel:

$$assert(tpl\_h\_prio \neq -1) \tag{1}$$
$$assert(tpl\_kern \neq NULL) \tag{2}$$
$$assert(tpl\_kern \rightarrow state == RUNNING) \tag{3}$$

tpl_h_prio is the value of the highest-priority task in the ready queue in the Trampoline kernel. $tpl\_h\_prio \neq -1$ is supposed to be true whenever rescheduling is necessary. tpl_kern stores the key information of the currently running task. $tpl\_kern \neq NULL$ and $tpl\_kern \rightarrow state == RUNNING$ checks if the state of the running task is *RUNNING* when the scheduler is called.

We performed verification of these three assertions using the model checker CBMC, Root-Level random testing, and End-Level random testing. The verification cost in terms of the number of verification conditions and the resource requirements was measured for CBMC verification. Branch coverage was measured using the Squish Coco code coverage tool [2] for random testing. All experiments were performed on Linux Fedora 16 OS, with Intel Xeon 3.4GHz e3-1270 processor and 32GB of 1333MHz DDR3 RAM.

Property	tpl_h_prio != -1 code size : 437 lines *3 End-Level Functions / 19 Functions in total*			tpl_kern != NULL tpl_kern->state == RUNNING code size : 787 lines *7 End-Level Functions / 32 Functions in total*		
Unwind	VCC	Time(s)	Memory(MB)	VCC	Time(s)	Memory(MB)
Unwind 3	15	2	55.72	53	15	200.94
Unwind 7	43	91	297.14	157	4,100	1288.90
Unwind 10	64	2,379	923.67	235	42,241	1942.54
Unwind 15	99	30,748	2693.75	365	> 6 days	> 7366.89

Figure 11: Time and memory space to verify with CBMC

Figure 11 shows the time and memory space it took to verify the Trampoline operating system with the End-Level Environment Model using the model checker CBMC. Time and memory space increase exponentially as the length of the End-Level function calls (unwind value) increases. CBMC verifies the assertions by searching through every possible scenario with the length of the unwind value, making it a powerful method. With the given resources, CBMC reported no counter examples up to the unwind value 10, but was not able to finish its verification process for the unwind value 15 after 6 days. We do not report the result of CBMC model checking using Root-Level environment models since it is too costly to perform even with the value of unwind option 10.

Property	tpl_h_prio != -1 code size : 1337 lines (8 Root-Level, 3 End-Level Functions) / 50 Functions					tpl_kern != NULL tpl_kern->state == RUNNING code size : 1378 lines (9 Root-Level, 7 End-Level Functions) / 52 Functions				
Length of the Scenario	14	20	22	32	34	14	20	22	32	34
tpl_schedule_from_running	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)
tpl_schedule_from_dying	-	-	-	-	-	80%(4/5)	80%(4/5)	60%(3/5)	80%(4/5)	100%(5/5)
tpl_schedule_from_waiting	-	-	-	-	-	0%(0/3)	100%(3/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)
tpl_start_scheduling	-	-	-	-	-	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)
tpl_wait_event_service	-	-	-	-	-	0%(0/5)	60%(3/5)	60%(3/5)	60%(3/5)	60%(3/5)
tpl_activate_task	-	-	-	-	-	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)
tpl_set_event	-	-	-	-	-	0%(0/5)	80%(4/5)	80%(4/5)	80%(4/5)	80%(4/5)
tpl_get_proc	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	-	-	-	-	-
tpl_put_new_proc	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(3/3)	66.67%(2/3)	-	-	-	-	-
Time(s)	-	-	-	-	-	-	-	-	-	-
Memory(MB)	2.64	2.64	2.64	2.64	2.64	2.64	2.64	2.64	2.64	2.64

Figure 12: Coverage, time, and memory space to verify with Root-Level Random Testing

Property	tpl_h_prio != -1 code size : 437 lines 3 End-Level Functions / 19 Functions in total					tpl_kern != NULL tpl_kern->state == RUNNING code size : 787 lines 7 End-Level Functions / 32 Functions in total				
Length of the Scenario	10	50	100	1000	10000	10	50	100	1000	10000
tpl_schedule_from_running	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	33.33%(1/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)
tpl_schedule_from_dying	-	-	-	-	-	60%(3/5)	60%(3/5)	100%(5/5)	100%(5/5)	100%(5/5)
tpl_schedule_from_waiting	-	-	-	-	-	0%(0/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)	66.67%(2/3)
tpl_start_scheduling	-	-	-	-	-	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)
tpl_wait_event_service	-	-	-	-	-	0%(0/5)	60%(3/5)	60%(3/5)	60%(3/5)	60%(3/5)
tpl_activate_task	-	-	-	-	-	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)	100%(4/4)
tpl_set_event	-	-	-	-	-	0%(0/5)	80%(4/5)	80%(4/5)	100%(5/5)	100%(5/5)
tpl_get_proc	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	100%(3/3)	-	-	-	-	-
tpl_put_new_proc	66.67%(2/3)	66.67%(2/3)	100%(3/3)	100%(3/3)	100%(3/3)	-	-	-	-	-
Time(s)	-	-	-	-	0.04	-	-	-	-	0.07
Memory(MB)	1.06	1.06	1.06	1.06	1.06	1.07	1.07	1.07	1.07	1.07

Figure 13: Coverage, time, and memory space to verify with End-Level Random Testing

There are a few dozens of extracted functions in random testing environments, including End-Level Functions, as illustrated in Figures 12 and 13. Due to a lack of space, only the test results of End-Level Functions are described. Root-Level Random Testing (Figure 12) is much faster (less than 1/100 seconds), consumes little memory (up to 2.64MB of memory), and achieves a certain level of test coverage quickly, but the coverage does not improve after test sequences of length 34. In End-Level Random Testing (Figure 13), a test sequence of length 100 achieves a certain level of coverage both for tpl_h_prio and tpl_kern. The coverage stays the same afterwards. End-Level Random Testing required around 1.06 1.07 MBytes of memory.

For many cases, the coverage did not increase even with lengthier test cases. There can be two reasons why some part of the code are unreachable. The first reason is exception handling; parts of the code for exception handling are never reached unless an exceptional situation occurs. The second reason is that some variables in conditional statements are not included in the Extended Verification Target Variable. So the behavior of updating these variables might not be fully extracted, which can make some conditional statements fixed. An example of this case is illustrated in Figure 14. This conditional statement is only

```
void tpl_schedule_from_running(void){
        ......
        // READY_AND_NEW
        if (tpl_kern.running->state == READY_AND_NEW)
        {
            tpl_init_proc(tpl_kern.running_id);
        }
        ......
}
```

Figure 14: Example of uncovered conditional statement

```
tpl_get_proc                 -> tpl_h_prio :  2, taskNum : 4, activationCount(T1, T2, T3) : 1, 255, 2
tpl_put_new_proc             -> tpl_h_prio :  2, taskNum : 5, activationCount(T1, T2, T3) : 1, 255, 1
---put_new_proc : 1
tpl_get_proc                 -> tpl_h_prio :  2, taskNum : 4, activationCount(T1, T2, T3) : 1, 0, 1
tpl_get_proc                 -> tpl_h_prio :  0, taskNum : 3, activationCount(T1, T2, T3) : 1, 0, 0
tpl_schedule_from_running  -> tpl_h_prio : -1, taskNum : 3, activationCount(T1, T2, T3) : 0, 0, 0
RandomTest: RandomTest.c:505: tpl_schedule_from_running: Assertion `tpl_h_prio != -1' failed.
```

Figure 15: Error caused by overflow

executed when *tpl_schedule_from_dying* or *tpl_activate_task* is executed before this code. But these two functions are out of boundary in this model because the model is generated only with regard to the property *tpl_h_prio != 0*. Thus coverage cannot be increased, and testing terminates.

In terms of comprehensiveness, CBMC is the most powerful method, verifying every possible scenario within the same length, but it is limited by the length. As shown in Figure 11, the verification cost increases exponentially as unwinding depth increases. Therefore, CBMC cannot detect potential faults that can be identified only in long task scenarios. Unlike verification with CBMC, the length of the scenarios is not limited in random testing since the cost is much cheaper than CBMC, as shown in Figure 13. Though it cannot be comprehensive, it can be more effective in stress testing, since the length of the test sequences can be sufficiently long.

End-Level Random Testing did in fact, catch some overflow errors in the Trampoline kernel as illustrated in Figure 15. These errors have occurred because the size of the variables saving the activation count is limited to 8 bits; the second line of Figure 15 shows that *Task*2 has the activation count 255, but adding another activation changed its value to 0. So Trampoline has changed the value of *tpl_h_prio* to -1, meaning that the process table has no activated task available. The variable size is implementation-specific and is not constrained to 8 bits, neither in the OIL specification nor in the OSEK/VDX specification. This problem could be addressed by constraining the size to 8 bits in the OIL specification.

A Model checker could find this type of potential faults if we can set the value of the unwind option larger than 255, but our experiments could not identify it due to resource limitations. End-Level Random Testing is appropriate for finding this kind of errors because the cost does not increase much even with lengthy test scenarios.

We could not identify the same fault using Root-Level Random Testing, either. The main reason is that Root-Level Random Testing is coupled with a pre-defined OIL configuration file. The OIL file

specifies the typical system configuration, and thus, activating a task over 255 times is not likely to happen unless we specifically aim at stress testing. End-Level Random Testing is more effective in stress testing, since it is not constrained by the system configuration and can test abnormal cases.

API-Level Random Testing, however, is beneficial in that it is not necessary to do additional API-Level analysis when testing identifies faults, which is necessary in model checking and End-Level Random Testing.

8 Conclusion

This paper presented methods and tools for environment generation and code abstraction to improve the efficiency of verification using model checking and testing. The effect of using the suggested approach was demonstrated through a series of experiments using the Trampoline operating system as a case example. The benefit of property-based environment generation is two-fold: (1) it reduces verification cost by reducing the target code and by limiting its environment to the task interaction scenario relevant to the verification property, and (2) it simplifies the analysis process and localizes the verification activity by focusing on the points of interest.

The experiments revealed relative pros and cons of the three verification methods and identified potential safety faults, which suggests the following collaborative use of model checking and testing;

1. Apply End-Level Random Testing first for stress testing.

2. Apply Root-Level Random Testing to conform the errors identified through End-Level Random Testing.

3. Apply model checking using CBMC last for comprehensive verification within a limited scope.

Our tool still needs some improvements. First, conditional dependencies need to be considered so that test coverage can be improved. Second, Root-Level scenario generation currently assumes a fixed OIL configuration. We would like to relax the condition so that an arbitrary OIL can be handled by the tool.

References

[1] *OSEK/VDX Portal*. Http://portal.osek-vdx.org.

[2] *Squish Coco Code Coverage*. Http://www.froglogic.com/squish/coco/.

[3] *Trampoline – OpenSource RTOS project*. Http://trampoline.rts-software.org.

[4] *Understand: Source Code Analysis and Metrics*. Http://www.scitools.com/.

[5] David Binkley (1999): *The Application of Program Slicing to Regression Testing*. *Information and Software Technology*, pp. 583–594, doi:10.1016/S0950-5849(98)00085-8.

[6] Manfred Broy (2006): *Challenges in automotive software engineering*. In: *Proceedings of the 28th International Conference on Software Engineering*, doi:10.1145/1134285.1134292.

[7] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti & Jacques Julliand (2012): *Program slicing enhances a verification technique combining static and dynamic analysis*. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pp. 1284–1291, doi:10.1145/2245276.2231980.

[8] Jiang Chen & Toshiaki Aoki (2011): *Conformance Testing for OSEK/VDX Operating System Using Model Checking*. In: *18th Asia-Pacific Software Engineering Conference*, doi:10.1109/APSEC.2011.26.

[9]   Edmund Clarke, Daniel Kroening & Flavio Lerda (2004): *A Tool for Checking ANSI-C Programs*. In: *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, doi:10.1007/978-3-540-24730-2_15.

[10] Edmund M. Clarke, Orna Grumberg & Doron Peled (1999): *Model Checking*. MIT Press.

[11] Matthew Dwyer & Corina Pasareanu (1998): *Filter-Based Model Checking of Partial Systems*. In: *6th ACM SIGSOFT International symposium on Foundations of Software Engineering*, pp. 189–202, doi:10.1145/288195.288307.

[12] Dimitra Giannakopoulou, Corina S. Pasareanu & Howard Barringer (2002): *Assumption Generation for Software Component Verification*. In: *17th IEEE International Conference on Automated Software Engineering*, pp. 3–12, doi:10.1109/ASE.2002.1114984.

[13] A. Gupta, K.L.McMillan & Z. Fu (2008): *Automated Assumption Generation for Compositional Verification*. *Formal Methods in System Design* 32, pp. 285–301, doi:10.1007/978-3-540-73368-3_45.

[14] Rajiv Gupta, Mary Jean Harrold & Mary Lou Soffa (1992): *An Approach to Regression Testing using Slicing*. In: *Proceedings of the Conference on Software Maintenance*, pp. 299–308, doi:10.1109/ICSM.1992.242531.

[15] Nannan He & Michael S. Hsiao (2007): *Bounded Model Checking of Embedded Software in Wireless Cognitive Radio Systems*. In: *25th International Conference on Computer Design*, pp. 19–24, doi:10.1109/ICCD.2007.4601875.

[16] Gerard J. Holzmann (2003): *The* SPIN *Model Checker: Primer and Reference Manual*. Addison-Wesley Publishing Company.

[17] Natalia Ioustinova & Natalia Sidorova (2002): *Abstraction and Flow Analysis for Model Checking Open Asynchronous Systems*. In: *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, doi:10.1109/APSEC.2002.1182992.

[18] Jürgen Mössinger (2010): *Software in Automotive Systems*. *IEEE Software* 27(2), pp. 92–94, doi:10.1109/MS.2010.55.

[19] Wonhong Nam, P. Madhusudan & Rajeev Alur (2008): *Automatic symbolic compositional verification by learning assumptions*. *Formal Methods in System Design*, doi:10.1007/s10703-008-0055-8.

[20] Pascal Raymond Nicolas Halbwachs, Fabienne Lagnier (1993): *Synchronous Observers and the Verification of Reactive Systems*. In: *Third International Conference on Algebraic Methodology and Software Technology, AMAST'93*.

[21] Pavel Parizek & Frantisek Plasil (2007): *Partial Verification of Software Components: Heuristics for Environment Construction*. In: *33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, doi:10.1109/EUROMICRO.2007.46.

[22] John Penix, Willem Visser, Seungjoon Park, Corina Pasareanu, Eric Engstrom, Aaron Larson & Nicholas Weininger (2005): *Verifying Time Partitioning in the DEOS Scheduling Kernel*. *Formal Methods in Systems Design Journal* 26(2), pp. 103–135, doi:10.1007/s10703-005-1490-4.

[23] O. Tkachuk, M.B. Dwyer & C.S. Pasareanu (2003): *Automated Environment Generation for Software Model Checking*. In: *18th IEEE International Conference on Automated Software Engineering*, pp. 116–129, doi:10.1109/ASE.2003.1240300.

[24] M. Weiser (1984): *Program Slicing*. *IEEE Transactions on Software Engineering* SE-10(4), pp. 352–357, doi:10.1109/TSE.1984.5010248.

[25] Kenro Yatake & Toshiaki Aoki (2010): *Automatic Generation of Model Checking Scripts based on Environment Modeling*. In: *17th International SPIN Conference on Software Model Checking*, doi:10.1007/978-3-642-16164-3_5.