Safety Properties based Scenario Generation for Model Checking Trampoline OS

Nahida Sultana Chowdhury[1] and Yunja Choi[2]

[1]*Australia's ICT Research Centre, University of New South Wales, Sydney 2032, Australia*
[2]*Department of Computer Science and Engineering, Kyungpook National University, Daegu 702-010, South Korea*
nahida_uap@yahoo.com

Abstract

Model checking has proven to be a successful technology to verify real-time embedded and safety-critical systems. However an application of model checking in practice still requires manual construction of an environment model, which has a direct impact on verification cost. This paper suggests an automated scenario generation technique through a property-based static analysis of function-call relationship of the program source code. We present the scenario generation process and show application results on the Trampoline operating system using CBMC as a back-end model checker. The experimental result shows that our approach is able to reduce the verification cost significantly in terms of memory space and run time.

Keywords: *Model checking, CBMC, Verification, Trampoline OS, OSEK/VDX, Scenario Generation*

1. Introduction

Modern cars consist of up to 100 of small electrical control units to improve the comfort and reliability of vehicles, which are controlled by real time operating systems. Therefore, the safety of the operating system is a pre-requisite to ensure the safety and reliability of an automotive system.

Model checking [1], is an effective technique to identify subtle issues in software safety this is particularly important for automotive systems. Current technical advances in model checking enables engineers directly applying the technique to program source code, removing the manual model construction process required formerly. CBMC [2] is one of such model checking tools, which is capable of verifying almost full ANSI C. It is capable of verifying 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 supports full set of ANSI-C.

An application of model checking in practice still requires manual construction of an environment model, which has a direct impact on verification cost. Especially when the technique is to be applied to embedded software, such as automotive real-time operating systems, we need to first define the interaction scenario between the operating system and application programs in general. A straightforward approach that models the interaction behavior as infinite and non-deterministic choices among the system APIs is too costly, since the model checking technique is based on the exhaustive search of the whole system state-space.

Our goal is to automatically construct an environment model that preserves environmental constraints and minimizes the amount of source code to be verified for a given property. Our work anticipates that the efficiency of model checking depends on the modeling of the application environment and suggests an approach to automatically generate the environment of an automotive operating system in terms of interaction scenarios between the operating system and its application environment. This paper presents our approach is automatically generate environment models from the structural data dependency information analyzed from the source code and experiments comparing the model checking performances using typical environment model and our suggested environment model data shows the efficiency of our approach over Trampoline OS by using CBMC. The tool structure has provided in Figure 1 to present the tool chain that covers the entire system development life cycle including scenario generation and verification module.

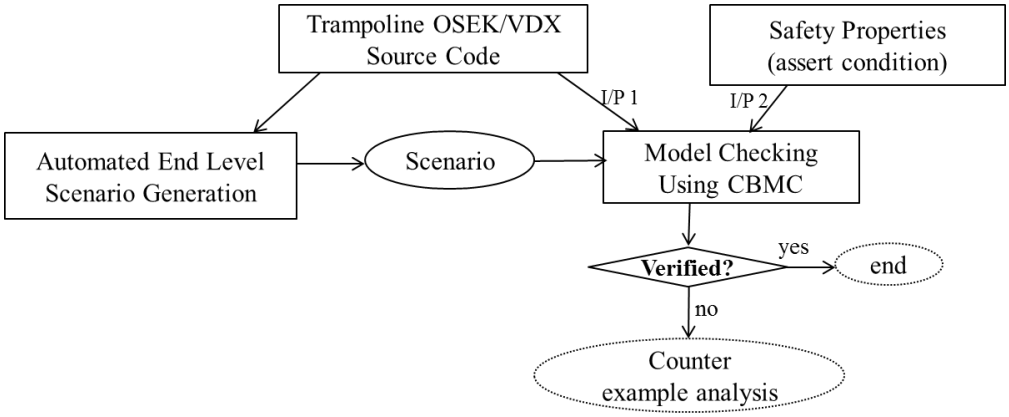

Figure 1. Scenario generation and verification tool chain

The remainder of this paper is organized as follows. Section 2 introduces the related literature in this domain; Section 3 contains motivation of our approach; Section 4 presents methods and process for the automated scenario generation technique. Experimental result and evaluation are displayed in Section 5 where Section 6 presents the limitations of the approach. Finally Section 7 concludes the paper with the discussion of further improvement on this development.

2. Related Literature

Most popular technique for scenario generation is UML-based scenario generation [3, 4]. Most of those approaches generate abstract test cases directly from the UML models, and none of them makes the use of the system source code during the scenario generation. In [4], they have presented an approach about automated scenario generation based on UML activity diagrams. But in this process they did not developed the verification process for the generated scenario. System model can be two types one is Platform Independent Model (PIM) and another is Platform Specific Model (PSM). Three categories of code generators can be identified depending on the degree of completeness of the PIM and of the resulting PSM: skeleton generation, partial generation, and full generation. Skeleton generation has been adopted by most tools, and deals only with the static structure of the system. Partial generation takes a more complete specification as input. Behavior may be modeled by state machines, sequence diagrams, activity diagrams, etc. However, these diagrams are most of the time incomplete at PIM level. Full generation tools introduce at PIM level a new action

language, conforming to action semantics. The Action Semantics proposal for our approach defines by calls and called-by relationship to allow an easy mapping of system structure.

Few works applies the model-based approaches to the development of automotive electronics system based on OSEK/VDX standard. In [5], SmartOSEK platform is to build a model-based development environment for automobile applications compliant with OSEK/VDX specifications. It consists of an operating system and an integrated development environment that consists of many convenient tools. In [6], they present model-based approach to develop automotive electronics software by SmartOSEK. Also they present simulator-based approach to verify the system model.

In [7] they describe a software analysis tool for the debugging and verification of TinyOS 2, MSP430 applications at compile-time using CBMC model checker. This tool is the first to allow the programmer to verify a TinyOS application statically; given assumptions about the behavior of the node's environment and assertions upon the state of the node itself, the tool explores all possible program executions and returns to the programmer an error trace leading to the violation of an assertion, if any exists. Besides memory related errors (out-of-bounds arrays, null-pointer dereferences), it also support application-specific assertions, including low-level assertions upon the state of the registers and peripherals. In this approach there concern was to find the all possible execution path and verify them with CBMC. And in our approach particular objective is, for each safety properties find the all possible scenarios and verify them to characterize the reliability of the system. Also in this work a TinyOS application is verified statically where in our process the scenario generation process and verification is automated.

Compared with the previous literatures in our approach, we have applied a different method to generate environment model of Trampoline OS. We did make use of the source code directly for scenario generation because there is no model available for trampoline OS but only code. Using Understand Source Code Analysis & Metrics [8], we have extracted the structural data dependency relations where data about called-by graphs and call graphs of functions from Trampoline source code. The generated scenario from called-by graphs and call graphs presents a valid calling sequence of the functions according to the source code structure. The CBMC tool is customized in our approach to make use of the automated scenario generation for verification. Most incentive point of our work is the whole process is automated and according our knowledge the technique is newly introduced which is effective for model checking process in terms of memory space usage and required runtime.

3. Motivation

Trampoline [9], is an open source operating system written in C and is based on OSEK/VDX; where OSEK/VDX [10] is an international standard for real-time operating system used in the field of automotive embedded software.

Correctness is a crucial concern for real-time operating system, because it affects the safety properties of the entire system. In embedded system the assert conditions are concerned for safety-critical properties, where CBMC uses bounded model checking techniques to verify certain properties such as the violation of assertions. It implements a technique called Bounded Model Checking (BMC), that transforms the program and property into Boolean formula and uses SAT solver to show whether the formula is satisfiable or not. So if any violated property exists then it will return a counterexample with tracing information, which confirms an ideal verification for the safety issues of embedded system.

An application of model checking in practice requires environment model of the system for verification, where the environment model based on the interaction scenario of the operating system and application environment. For particular properties such as safety properties case

we need to generate the environment model from the source code based on the properties structural dependency relation and then the environment model can pass to the model checker for safety properties verification (Figure 2).

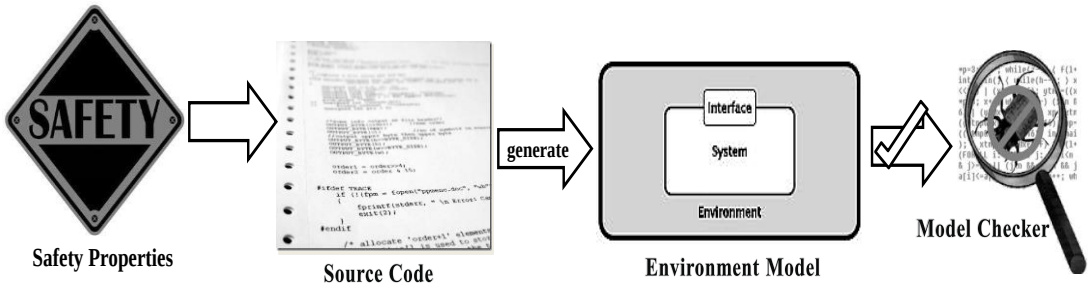

Figure 2. System safety properties verify by model checker

Based on the verification environment, the existences industry-standard technologies are constrained random verification, assert based verification, and functional coverage-driven verification. To develop an environment scenario for verification a straightforward way is to follow the constrained random verification because it increases the reliability in verification by detecting the bugs in corner cases. Here the environment scenario is based on random system-level APIs. However model checking under arbitrary environment can lead to negative outcomes. It can be costly verification; a randomly generated API services scenario exercise the all possible sequence and such an environment exercises all possible interactions between the user task and the underlying operating system even when most of the interactions are not relevant to verify a given target property. Further it can include invalid scenarios; without knowing the structural dependency of a system randomly scenario generation process can produce the invalid scenarios.

To avoid the limitations of existence scenario generation techniques we need a better way to model the environment. Therefore we suggest a method based on four important aspects, which are:

1) From the arbitrary Scenario of API services, end-level-functions scenario has to be generated.
2) Constraints of API services have to be imposed in functions level scenario.
3) Structural dependency of functions need to be considered
4) Combindly use of constrained random verification and assert based verification technologies to generate the safety properties based scenario.

4. Proposed Scenario Generation Technique

Our approach automatically constructs an environment model by analyzing the structural information of source code and imposing external constraints identified from The OSEK/VDX standard. It is a property-based approach: given a property to be verified, we first identify variables and functions directly related to the property. We then extract call sequence related to the identified functions from which non-deterministic environment model is constructed. This section explains the details of our approach, starting from some definitions we use throughout this paper.

4.1. Terminology

Definition 1 A Generic-Model-Scenario (GM) is an arbitrary sequence of API calls of an operating system.

Definition 2 A Verification Target Variable (VT) is a global variable appeared in the property specification.

Definition 3 End-Level-Functions (ELF) is the set of functions which directly modify, set, or use elements in VT.

Definition 4 An End-level-functions-Scenario is the calling sequence of End-Level-functions with called-by structural constraints.

Definition 5 Root-Level-Functions are API services which are starting functions of the called-by graph of an end-level-function.

Definition 6 An Root-level-function-Scenario is the arbitrary sequence of Root-Level-Functions.

4.2. Property-based Scenario Generation

The approach is to generate the End-Level-Functions scenario for a *VT* by analyzing called-by graphs and call graphs of *ELF* from the kernel of the Trampoline operating system. The work flow diagram of our scenario generation process is presented in Figure 3.

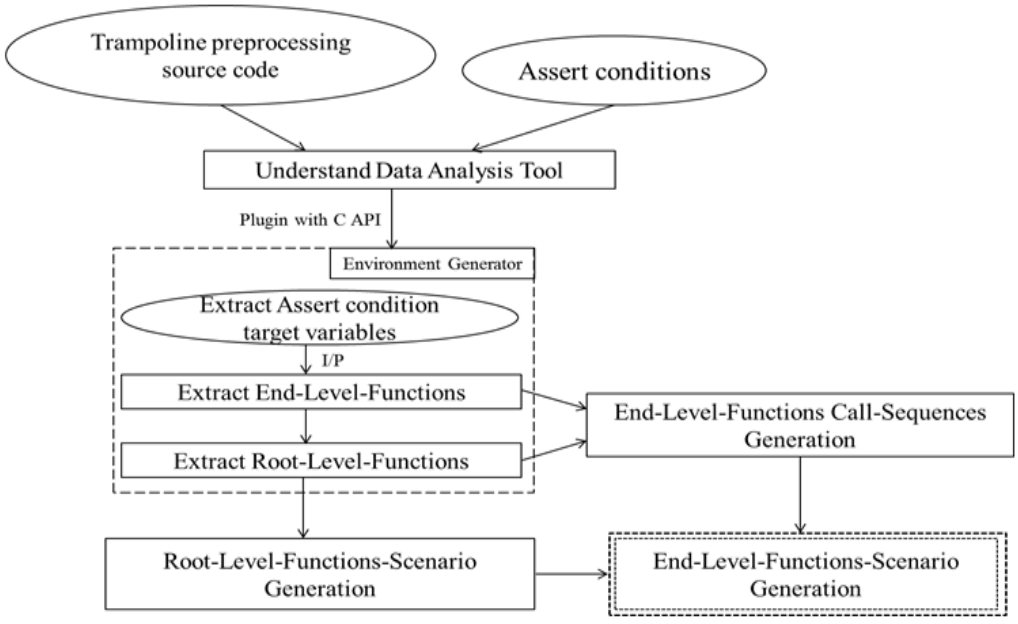

Figure 3. Workflow diagram of the scenario generation process

Using preprocessing source code of Trampoline OS, and assert condition properties database is created with the help of analysis tool. After that database is plugged in with C

API, where according to the one of extracted assert condition target variables End-Level-Functions-Scenario is generated. Finally using model checker verification is taken place for the extracted scenario from the Trampoline OS environment. To extract the data about called-by graphs and call graphs of functions from the Trampoline source code, Understand Source code analysis & Metrics tool is used in our work.

Property-based scenario generation is distributed into three parts, which are (1) Extraction of relevant Root-Level-Functions, (2) Pruning call sequences from the identified Root-Level-Functions to *ELF*, and (3) Non-deterministic choice of the pruned call sequences after applying constraints imposed by the OSEK/VDX standard.

4.2.1. Property-based Scenario Generation

For extracting the property based Root-Level-Functions the **first stage** is to find all *VT* and for a particular one variable then **second stage** is find *ELF*. Using Understand API to extract *ELF* from the system we have established two criteria. First one, if the *VT* is set by local variable or constant and secondly if the *VT* is set by other global variable of the system. If the *VT* is assigned by any local variable or constant of the system then only find *ELF* that have directly updated the value of that variable or use that variable in assert condition. In Figure 4 the classification of *ELF* references has shown where dot line represents the counted *ELF* types in our process. Again if the *VT* is set by other global variable in that case we need to consider the *ELF* list of that global variable also.

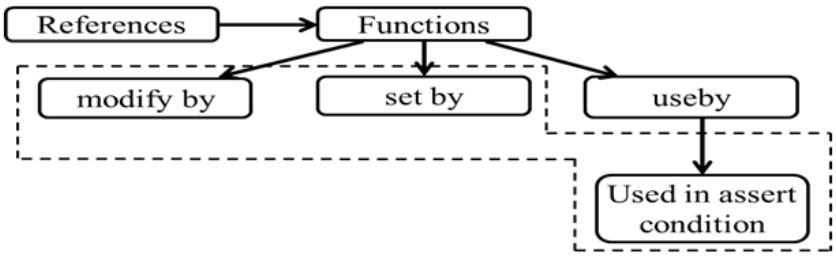

Figure 4. Classification of *ELF* references

And final stage is for the extracted *ELF* find the all possible Root-Level-Functions. For each *ELF* reference, we subsequently find out all possible Root-Level-Functions using called-by graph feature of Understand tool.

4.2.2. Pruning Call Sequences from Root-Level-Functions

To extract the calling sequence of *ELF*, need to know the system call structure. Therefore traverse from the Root-Level-Functions using call graph feature of Understand tool and find the calling sequence of the *ELF* through each root.

4.2.3. Scenario Generation with Imposing Constraints

In our approach, the system calling sequence of End-Level-Functions is known as scenario. Afterwards, in non-deterministic way chooses the Root-Level-Functions and sequentially generate the End-Level-Function's sequence. To make a valid scenario, we also need to consider the restrictions between two root functions. OSEK/VDX standard imposes several restrictions on services as shown in Table 1.

Table We have to consider these constraints to generate valid scenario. However in our approach the constraints part has established manually, for each system initially constraints need to develop manually after that it doesn't matter how many times we run the scenario generation technique it will be automatically imposed the developed constraint part to the scenario generation process. In our experiment, after studying the trampoline OS source code and the OSEK/VDX standard, we have collected the existing constraints.

Table 1. Constraints on API calls

1. A task must not be in a waiting state while holding resources, i.e. task cannot transit to *WaitEvent* from the *GetResource* state.
2. Without rescheduling Task other API services can be called between *GetResource* and *ReleaseResource* state, i.e. task cannot transit to *Schedule* state from the *GetResource* state.
3. A task must not terminate or chain another task while holding resources, i.e. a task cannot transit to *TerminateTask* or *ChainTask* from *GetResource* state.
4. Every task function should terminate using *TerminateTask* or *ChainTask.*
5. *StartOS* can be called in initially only.
6. Number of activate task function cannot be crossed the E_OS_LIMIT of Trampoline OS.
7. For a task function after calling *TerminateTask*, no API service will be allowed to call for that Task.

Without the constraint development module other modules in our process is fully automated and platform independent. Now the pseudo code for generating automated scenario of Trampoline OS is represented below:

V: Set of variables used in Safety Properties.
Dep_V: Set of dependent variables of v, where v∈V.
Dep_fn: Set of end-level-functions of v and dv, where v∈V and dv∈Dep_V.
Root_fn: Set of Root-level-functions of df, where df∈Dep_fn.
Seq_Root_fn: Arbitrary Sequence of Root_fn considering system constraints.
Seq_Dep_fn: Sequence of Dep_fn for each element of Seq_Root_fn.
RAND: is a function, takes the set of Root_fn and randomly returns an element of Root_fn.
Seq_Root_fn_constraint: is a function which takes the root-level-function name as input then return 0 if there has no constraints, otherwise return 1 if constraints exist.
Seq_fn: is a function, take the Root-level-function as input and return the corresponding End-level-functions sequence.

```
1.  void main( )
2.  {
3.      for (i=1 to a) // a is the total number of asserts condition
4.      Store asserts condition target variables in the set V;
5.      Input: Chose an element of V.
6.      Find Dep_V for input.
7.      Find Dep_fn for each element of Dep_V and v; where dfi≠dfj and
        dfi & dfj ∈ Dep_fn.
8.      Find Root_fn for each element of Dep_fn; where rfi≠rfj and
        rfi & rfj ∈ Root_fn.
9.      for (i=1 to n) //n is the length of scenario
10.     {
11.         Root_fn_name= RAND(Root_fn);
12.         if (Seq_Root_fn_constrain(Root_fn_name)==0)
13.             Don't store in the set Seq_Root_fn;
14.         else
15.             Store in the set Seq_Root_fn;
16.     }//for i
17.     for each i, where i ∈ Seq_Root_fn
18.     call Seq_fn(i) and store the end-level-functions sequence in the set
        Seq_Dep_fn.
19.     Verify the set Seq_Dep_fn with CBMC.
20. }
```

Figure 5. Pseudo code of the scenario generation process

5. Experiment Results

This section describes the verification result using CBMC and the generated scenarios. All experiments were conducted on an Intel Core2 Quad CPU Q8200, 2.33GHz server with 3GB of RAM running 64bit Widows 7 OS. From Trampoline OS we have chosen six safety properties for the verification, as shown in Table 2. Six variables are extracted from the properties, which are 'tpl_h_prio', 'tpl_fifo_rw', 'tpl_ready_list', 'tpl_kern', 'prio', 'tpl_locking_depth'.

Table 2. List of Safety Properties in Trampoline OS

1. assert((tpl_h_prio >= 0) && (tpl_h_prio <3))
2. assert (tpl_h_prio != -1);
3. assert (tpl_fifo_rw[tpl_h_prio].size > 0);
4. assert (tpl_fifo_rw[prio].size < tpl_ready_list[prio].size);
5. assert (tpl_kern.running != NULL);
6. assert (tpl_kern.running->state == RUNNING);
7. assert ((prio >= 0) && (prio < 3));
8. assert(tpl_locking_depth > =0);

In Figure 6 one of the small scenario for 'tpl_h_prio' is represented, which results in a successful verification by CBMC. This scenario contains the ELF sequence according to the randomly picked 10 root function references.

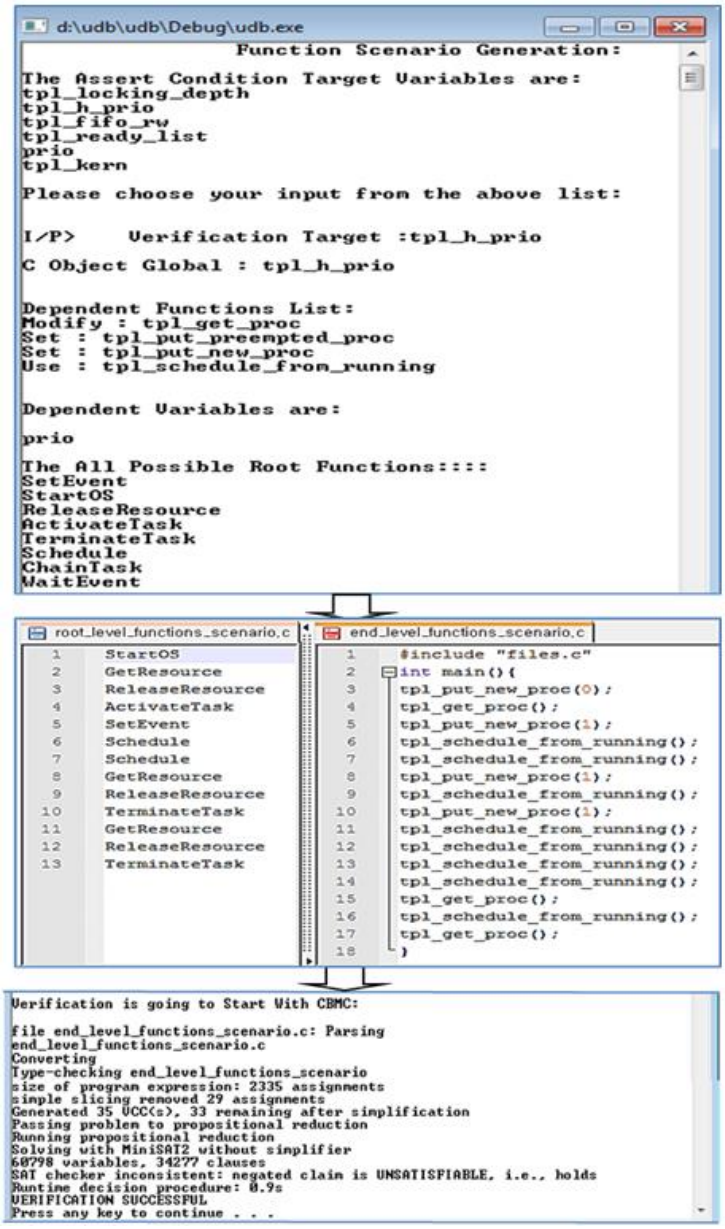

Figure 5. A small scenario of 'tpl_h_prio'

For all assertion in Trampoline OS verification has conducted using different scenario generation approaches which are typical used scenarios (root-level-functions-scenario and *GM*) and our proposed scenario (end-level-functions-scenario). For 'tpl_h_prio' verification target, the experimental data is presented in Table 3 for different length of scenario using different scenario generation approach. Table 3 covers the name of different scenario generation approach, the length of scenario, size of program expression (in term of assignments) and runtime (in sec). This resulted data shows that our proposed end-level-functions-scenario is able to reduce the memory space up to 38% and runtime up to 82% compare with other scenario generation technique.

Table 1. Run time data of different scenarios

Scenario Generation Approach	Length of Scenario	Size of Program expression (No. of assignments)	Runtime (s)
Root-Level-Functions-Scenario	10	24214	18.34
	25	30100	35.38
	50	41540	73.87
	100	60622	225.761
	500	**204458**	**Out of Memory**
Generic-Model-Scenario	10	21240	14.64
	25	22338	16.43
	50	29409	36.01
	100	37626	60.72
	500	**101653**	**814.49**
End-Level-Functions-Scenario	10	1853	0.77
	25	4198	1.44
	50	7178	2.47
	100	13853	5.97
	500	**63583**	**144.81**

Here Table 4 is presented the run time properties (Number of generated verification condition, Size of program expression and the Runtime) based on different asserts conditions of OSEK/VDX OS and number of root-level-functions has called. Column 1 contains the assert conditions, column 2 contains the assert condition target variable for each entity of column 1, column 3 presents the length of scenario in terms of number of root-level-functions, column 4 present the run time (in second), column 5 presents the number of verification condition generated in CBMC and column 6 presents the size of program expression (in number of assignments). For each assert condition target two test case results has provided, according to the length of scenario 500 and 1000. However in our approach scenarios can be generated infinitely but here we have limit the length until 500 and 1000 due to CPU memory space constraints.

Table 2. Run time data in verification time

Assert Conditions	Assert Condition Target Variable	Length of Scenario	Runtime (s)	No. of Generated VCC	Size of Program Expression (No. of assignments)
assert(tpl_kern.running!=NULL)	tpl_kern	500	567.232	520	**121913**
assert((tpl_kern.running->state)==RUNNING)		1000	**Out of Memory**	1096	**249792**
assert((tpl_h_prio >= 0) && (tpl_h_prio <3))	tpl_h_prio	500	203.267	690	76899
assert (tpl_h_prio != -1)		1000	892.319	1369	151877
assert (tpl_fifo_rw[tpl_h_prio].size > 0)	tpl_fifo_rw	500	11.986	838	21856
assert (tpl_fifo_rw[prio].size < tpl_ready_list[prio		1000	34.756	1669	43202
	tpl_ready_list	500	10.47	413	21005
		1000	34.035	849	43092

6. Limitations

For hardware issues (*i.e.*, '_setjmp', '_longjmp') hardware simulation is required to fetch the proper behavior. However Understand tool was unable to simulate the hardware behavior which resulted invalid call graph for '_setjmp', and '_longjmp' functions. Then trampoline behavior has carefully observed and manually set the activities of '_longjmp' and '_setjmp' functions. For example for the below case (Figure 7) 'StartOS' first call the 'tpl_init_machine' and then after finishing the all function calling of 'tpl_init_machine' it will call 'tpl_start_os'.

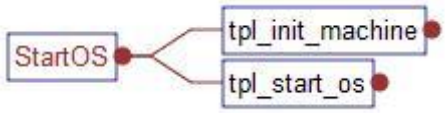

Figure 6. Call Graph of 'StartOS' (using Understand tool)

However in trampoline it's not work according to the Figure 7. Because in callee function's list of 'tpl_init_machine' there has a function, 'tpl_create_context_boot', which use '_longjmp' function reason to switch to 'tpl_start_os' function without finishing 'tpl_init_machine'. For this reason we have modified our source code manually to acquire the proper call sequence. In Trampoline source code, '_longjmp' and '_setjmp' has used different times and as we know the behavior of '_longjmp' and '_setjmp' is dynamic and depend on hardware functionalities. Therefore we were unable to deal with all the cases in our experiments.

7. Conclusion

Automotive embedded real time OS are often large and complex. That's why Model checking often suffers from the state space explosion problem when we intend to verify large scenario. In this paper we have represented an innovative and efficient method to generate the valid scenario for model checker in automated way, which according to our knowledge no such research work developed yet in this Trampoline OS domain. Most importantly the technique has shown its efficiency in terms of runtime and usage memory space compares with others literatures in this issue. The significant key facts are:

a) Scenarios are generated referring to the end-level-functions call sequence (call graph and called by graph);

b) Only valid scenarios are generated;

c) The scenario generation is performed considering the constraints imposed by international standard OSEK/VDX;

d) The last and the most importantly, without deep knowledge about the source code we can easily generate the valid scenario automatically, which will provide the opportunity to remove the time constraint and will allow to easily handle the source code.

In future work we intend to expand this work to make it more acceptable. With this strategy we are planning to focus on the below issues:

- Our main focus is to generate an alternate way to fetch the hardware behavior, which can help to handle the '_longjmp' and '_setjmp' issues.
- Need to give more attention on constraints part to make sure that all are accounted for.
- Also it needs more experiment with other source code suit to assure its usability and efficiency.
- Lastly, different model checker such as SatAbs, FeaVer, can be used in the verification process instead of CBMC. So we can make a comparison about features and effectiveness between different model checkers for this approach.

Acknowledgements

"This work was supported by the IT R&D program of MKE/KEIT. [10041145, Self-Organized Software-platform (SOS) for welfare devices]".

References

[1] M. Muller-Olm, D. Schmidt and B. Steffen, "Model Checking: A Tutorial Introduction," SAS'99, LNCS 1694, **(1999)**, pp. 330-354.
[2] E. Clarke, D. Kroening and F. Lerda, "A tool for checking ANSI-C programs. In: K. Jensen, A. Podelski (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)," Lecture Notes in Computer Science, vol. 2988, Springer, Berlin, **(2004)**, pp. 168-176.
[3] D. Latella and M. Massink, "On testing and conformance relations for UML statechart diagram behaviors," Proc. ACM SIGSOFT Int. Symp, Software Testing and Analysis, Roma, Italy, ACM Software Eng. Notes, vol. 27, **(2002)** July, pp. 144–153.
[4] P. G. Sapna and H. Mohanty, "Automated Scenario Generation based on UML Activity Diagrams," International Conference on Information Technology, **(2008)**, pp. 209-214.
[5] M. Zhao, Z. Wu, G. Yang, L. Wang and W. Chen, "SmartOSEK: A Dependable Platform for Automobile Electronics," The First International Conference on Embedded Software and System, vol. Springer-Verlag GmbH ISSN: 0302-9743, **(2004)**, pp. 437.
[6] G. Yang, M. Zhao, L. Wang and Z. Wu, "Model-based Design and Verification of Automotive Electronics Compliant with OSEK/VDX," Proceedings of the Second International Conference on Embedded Software and Systems, **(2005)**.
[7] D. Bucur and M. Kwiatkowska, "On Software Verification for Sensor Nodes," Journal of Systems and Software table of contents archive, vol. 84, Issue 10, **(2011)** October, pp. 1693-1707.
[8] "Understand Source code analysis and Metrics," http://scitools.com/index.php.
[9] R.-T. S. group IRCCyN, Trampoline, http://trampoline.rts-software.org/.
[10] OSEK Group, http://www.osek-vdx.org.
[11] "CBMC Manual", http://www.cprover.org/cprover-manual/cbmc.shtml.
[12] "OSEK/VDX Oerating System, Version 2.2.3", http://portal.osek-dx.org/files/pdf/specs/os223.pdf.
[13] "OSEK realtime operating system – osCAN Technical Reference", http://www.vector.com.
[14] "Trampoline OS Source Code Download", http://trampoline.rts-software.org/spip.php?rubrique3.
[15] "Trampoline OS Documentation", http://trampoline.rts-software.org/spip.php?rubrique1.

Authors

Nahida Sultana Chowdhury is a Research student in Australia's ICT Research Centre, University of New South Wales, Sydney, Australia. She completed her Master's in Computer Science and Engineering from Kyungpook National University, South Korea, in August 2012. She has obtained the B.Sc. degree in Computer Science and Engineering from the University of Asia Pacific, Bangladesh, in April, 2008.

Yunja Choi has been serving as an Assistant professor of the Department of Computer Science and Engineering in Kyungpook National University, Daegu, South Korea.