Scenario Generation for Model Checking Operating Systems

Nahida Sultana Chowdhury and Yunja Choi

Dept. Of Computer Science and Engineering, Kyungpook National University, South Korea
nahida_uap@yahoo.com, yuchoi76@knu.ac.kr

Abstract. 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.

Keywords: Trampoline OS, CBMC, Verification, Scenario Generation.

1 Introduction

Model checking enables engineers directly apply the technique to program source code removing the model construction process required formerly. 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. This paper presents our approach which automatically generates environment models using the structural data dependency information analyzed from the source code. The experimental result shows the efficiency of our approach using the Trampoline operating system as a case example.

OSEK/VDX [1] is an international standard for real-time operating system used in the field of automotive embedded software. Trampoline is an open source operating system written in C and is based on OSEK/VDX. In embedded system safety properties can be specified as assert conditions and CBMC [2] use bounded model checking techniques to verify the assertions. If any violated property exists then it returns a counterexample with tracing information, which provides useful information for safety analysis. CBMC requires specifying environments for the verification, application scenario and the validity of the environment model which have a big impact on the efficiency of model checking. When model checking is applied under arbitrary environment it will exhaustively exercise all possible call sequences and its verification result will contain false-negatives, impossible counterexamples. We note that if we look at the Trampoline source code the valid call sequence can be identified. Therefore we suggest a method to automatically generate valid application scenarios for the system.

2 Property-based Scenario Generation Approach

In our approach, scenarios are generated through the analysis of function's *called-by graphs* and *call graphs* of the program source code. Property-based scenario generation consists of three parts: (1) extraction of relevant Root-Level-Functions, (2) pruning call sequences from the identified Root-Level-Functions to End-Level-Functions, and (3) non-deterministic choice of the pruned call sequences after applying constraints imposed by the OSEK/VDX standard. In the first step, starting from variables participating in the target verification property, we first extract End-Level-Functions, which directly modify or use the variables. The end-level-functions are then traced in terms of function-call dependency up to the Root-Level-Functions, which are API services provided by the operating system. Since the identified Root-level-functions may also include function calls that do not lead to the end-level-functions, the second part eliminates the irrelevant paths from the root-level-functions. Finally, the third part generates environment model that exercises all possible call sequences from the identified root-level-functions to the end-level-functions, anticipating only regal scenarios that obey requirements specified in the international standard OSEK/VDX. This approach is implemented on top of Understand analysis tool [3]. The tool structure has provided in Figure 1 to present how each module of our experiment link with each other.

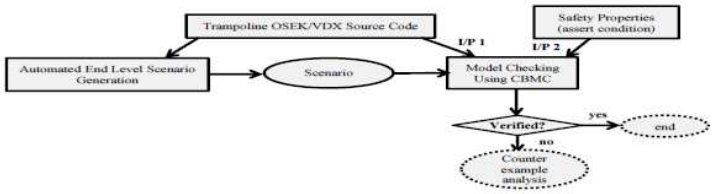

Figure 1: Scenario generation and verification tool chain

In our approach, the system calling sequence of *End-Level-Functions* is known as a scenario. Afterwards, in non-deterministic way we choose the *Root-Level-Functions* and sequentially generate the *End-Level-Function* sequence. To make a valid scenario, we also consider the constraints between two root functions.

3 Experiments

From Trampoline OS we have chosen six safety properties for the verification, as shown in Table 1. Six variables are extracted from the properties, which are 'tpl_h_prio', 'tpl_fifo_rw', 'tpl_ready_list', 'tpl_kern', 'prio', 'tpl_locking_depth'. For the all assertion in Trampoline OS we have conducted the scenario verification for end-level-functions sequence.

Table 1. List of safety properties in Trampoline OS

Assert Conditions
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);

Table 2 represents the verification run time properties based on different assert conditions of Tampoline OS and length of scenario (number of root-level-functions).

Table 2. Run time data in verification time

Assert Conditions	Assert Condition Target Variable	Length of Scenario (No. of Root Level	Runtime (s)	No. of Generated VCC	Size of Program Expression (No. of
assert(tpl_kern.running!=NULL)	tpl_kern	500	567.232	520	**121913**
assert((tpl_kern.running-		1000	**OM**	1096	**249792**
assert((tpl_h_prio >= 0) &&	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 >	tpl_fifo_rw	500	11.986	838	21856
		1000	34.756	1669	43202
assert (tpl_fifo_rw[prio].size < tpl_ready_list[prio].size)	tpl_ready_list	500	10.47	413	21005
		1000	34.035	849	43092

4 Conclusion

The important key facts are: (a) Scenarios are generated by analyzing the end level function 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.

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

References

1. OSEK/VDX Operating System Specification. http://www.osek-vdx.org.
2. CBMC Installation. http://www.cprover.org/cbmc/.
3. Understand Source code analysis and Metrics. http://scitools.com/index.php.