Automated Scenario Generation for Model Checking Trampoline Operating System

Nahida Sultana Chowdhury and Yunja Choi
Dept. of Computer Science and Engineering, Kyungpook National University
e-mail : nahida_uap@yahoo.com

Abstract

A valid scenario generation is essential for model checking software. This paper suggests an automated scenario generation technique through the analysis of function *called-by graphs* and *call graphs* of the program source code. We provide the verification process including the scenario generation and show application results on the Trampoline operating system using CBMC as a back-end model checker.

1. Introduction

CBMC [1] is one of bounded C code model checker. It is capable of verifying almost full ANSI C. It's a part of the CPROVER tool suite for the formal verification of both hardware and software designs. 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 advantages are: it is completely automated and supports full set of ANSI-C.

For embedded system most important concern is safety properties such as pointers and arrays and violation of assert condition. To verify the assert condition in ANSI-C code is the common issue for safety-critical embedded systems, where CBMC is fully capable to verify the assert condition whether it's satisfiable or not. So, currently CBMC is considered as a promising tool for analyzing embedded software and improving its quality.

However, the efficiency of model checking depends on the usage scenario of the application because the dependency among functions can be a contributing factor for correctly prove or disprove a given property. In this regard, scenario generation gains importance which is automated in our work.

This paper discusses about the approach to automate the scenario generation process using the Trampoline operating system as an example and CBMC as a model checker for the verification of assert conditions. In this paper, Section 2 presents the related work, section 3 presents motivation of this work, Section 4 presents methods and process for the automated scenario generation. Experimental result is displayed in Section 5 and Section 6 and Section 7 provide the limitation and future planning on this development.

2. Related Works

Most popular technique for scenario generation is UML model-based scenario generation [2, 3]. Most of those approaches generate abstract test cases directly from the UML models, and none of them makes the use of the programs during the scenario generation. In [3], 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.

Few works applies the model-based approaches to the development of automotive electronics system based on OSEK/VDX standard. In [4], 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 [5], they present model-based approach to develop automotive electronics software by SmartOSEK. Also they present simulator-based approach to verify the system model, which helps the developers to find the design fault and redesign the system model at early design time.

In our approach, we have applied a different method to generate scenario. 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 [6], we have extracted the 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. Most incentive point of our work is the whole process is automated.

3. Motivation

OSEK/VDX [7] is an international standard for real-time operating system used in the field of automotive embedded software. Trampoline [8], is an open source operating system written in C and is based on OSEK/VDX.

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 the violation of assertions. It implements a technique called Bounded Model Checking (BMC), where it's transformed the program and property into Boolean formula and SAT solver is used to show whether the formula is satisfiable or not. So if any violated property is exists than it will return a counterexample with tracing information, which confirmed an ideal verification for the safety issues of embedded system.

However, model checking exercises all possible scenarios for exhaustive verification, which often include invalid scenarios. Using CBMC under invalid application scenario

can make a reason for the violation of properties even though they are valid in fact. For example, the below code presents *tpl_get_proc* function which includes two assert conditions.

```
typedef signed char s8;
typedef unsigned char u8;
typedef s8 tpl_proc_id;
signed char tpl_h_prio = -1;
typedef struct tpl_priority_level;
typedef struct tpl_fifo_state;

tpl_fifo_state tpl_fifo_rw[3] ;
const tpl_priority_level tpl_ready_list[3] ;

static tpl_proc_id tpl_get_proc(void){
  /*function body*/
  assert((tpl_h_prio >= 0) && (tpl_h_prio < 3));
  assert(tpl_fifo_rw[tpl_h_prio].size > 0);
  /*function body*/
}
```

In the above example, the value of *tpl_h_prio* is initially -1 so "*assert ((tpl_h_prio >= 0) && (tpl_h_prio < 3))*" will be violated if the *tpl_get_proc* function is called first in a sequence. This can be occurred by arbitrary choosing *tpl_get_proc* function without knowing the valid calling sequence. If we analyze the source code using the function call sequence, *tpl_put_new_proc* function is supposed to be called before *tpl_get_proc* . The assert condition is true if the scenario preserves the valid call sequence. Because *tpl_put_new_proc* has modified the value of *tpl_h_prio*, code for *tpl_put_new_proc* function has shown below.

```
void tpl_put_new_proc(const tpl_proc_id proc_id){
  /*function body*/
  tpl_h_prio = prio; //prio value is 0 or more
  /*function body*/
}
```

Therefore, it is important to find out the valid scenario for more efficient application of model checking techniques to source code.

4. Scenario Generation

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. Understand databases can be accessed by using Perl or C/C++ API.

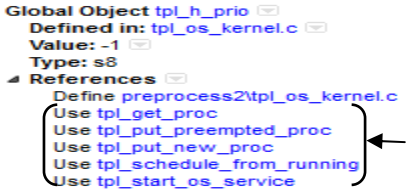
The approach is to generate scenario by analyzing *called-by* graphs and *call* graphs of functions used in trampoline operating system, which helps to create valid scenario. The work flow diagram of our scenario generation process is presented in Figure 3, considering only global objects.

4.1 Input Object

In our process input will be a global variable (pseudo code in figure 5, line 2), which is used in existing assert condition of trampoline OS. Local or public variables are not considered because that modifying a value of a local or public will not have an impact on other functions.

4.2 Finding Function Reference

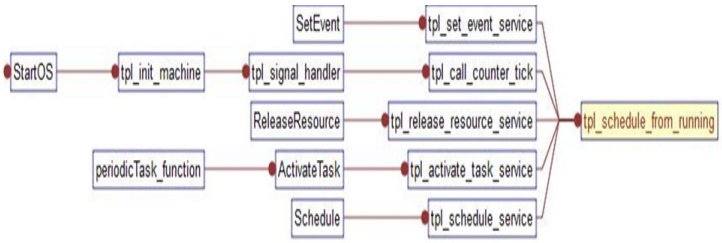
Function references are required to know for which functions are modifying or updating the value of this target object. Here, figure 1 represents an example on **Function references of variable** (pseudo code in figure 5, line 3, 4) using Understand API, which is specifying those functions who use that particular variable.



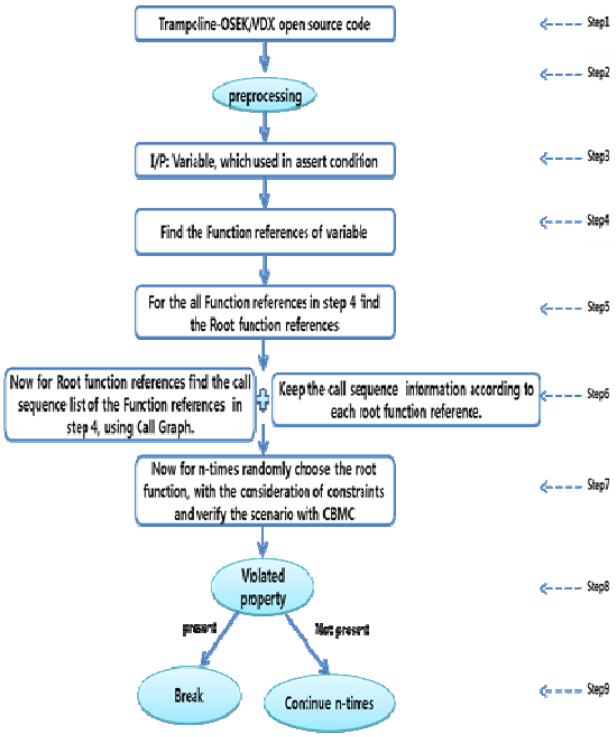
(Figure 1) Function Reference of 'tpl_h_prio' extracted using Understand API

4.3 Finding Root Function Reference

Root function references help to get the calling sequence of a function reference. From a function reference we subsequently find out all possible root functions who call these referenced functions. These possible root functions are known as **Root function references** (see pseudo code in figure 5, line 5, 6). Figure 2 represents the called-by graph of the *tpl_schedule_from_running* function, a function reference of tpl_h_prio (in figure 1). In figure 2 the root functions references are *SetEvent*, *StartOS*, *ReleaseResource*, *periodicTask_function*, *Schedule*. This way we find out root functions for each function reference.



(Figure 2) Demonstrating Root Function Reference for *tpl_schedule_from_running*, using Understand API

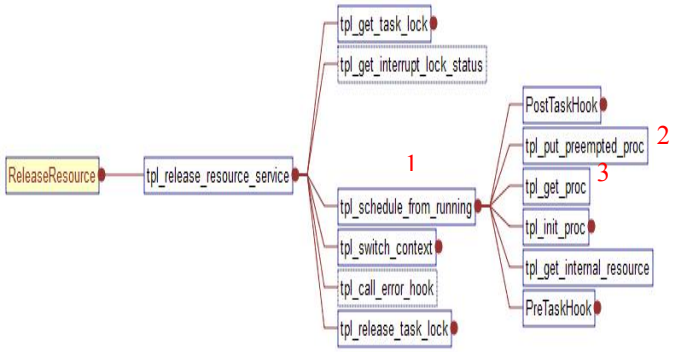


(Figure 3) Workflow diagram of the scenario generation process

4.4 To Generate Valid Scenarios

To know a call sequence of function references, it is important to derive the call sequence of functions, which is known as a scenario. We traverse from the root using *call graph* and find the calling sequence of the function reference through each root (pseudo code in figure 5, line 9-19). Afterwards, we randomly choose root functions one by one to call this function sequence arbitrarily.

For example the call sequence of function references from *ReleaseResource* root is "*tpl_schedule_from_running -> tpl_put_preempted_proc -> tpl_get_proc*" (using call- graph of Function ReleaseResource, which is showing in figure 4).



(Figure 4) Demonstrating Calling Sequence from ReleaseResource, using Understand API

4.5 Imposing Constraints

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. We have to consider these constraints to generate valid scenario (pseudo code in figure 5, line 23-32). In our experiment, after studying the trampoline OS source code and the OSEK/VDX standard, we have collected the existing constraints and handled them manually. For example, in figure 2, *schedule* function is not supposed to be called before *ReleaseResource* function if *GetResource* is called earlier. Because here *schedule* function is a rescheduling task (support the 2nd constraint of table 1).

Constraint's List
1. After TerminateTask, no task will be allowed to call.
2.Without rescheduling Task (i.e. TerminateTask, ChainTask, WaitEvent, Schedule) other API services can be called between GetResource and ReleaseResource.
3.EnableAllInterrupts will be allowed if before that DisableAllInterrupts has called and no API service will be allowed between this two API services.
4.CancelAlarm will be allowed if before that SetRelAlarm/ SetAbsAlarm has called.

(Table 1) Some of the Constraints are representing here

Now the pseudo code for generating automated scenario of Trampoline OS is represented in Figure 5.

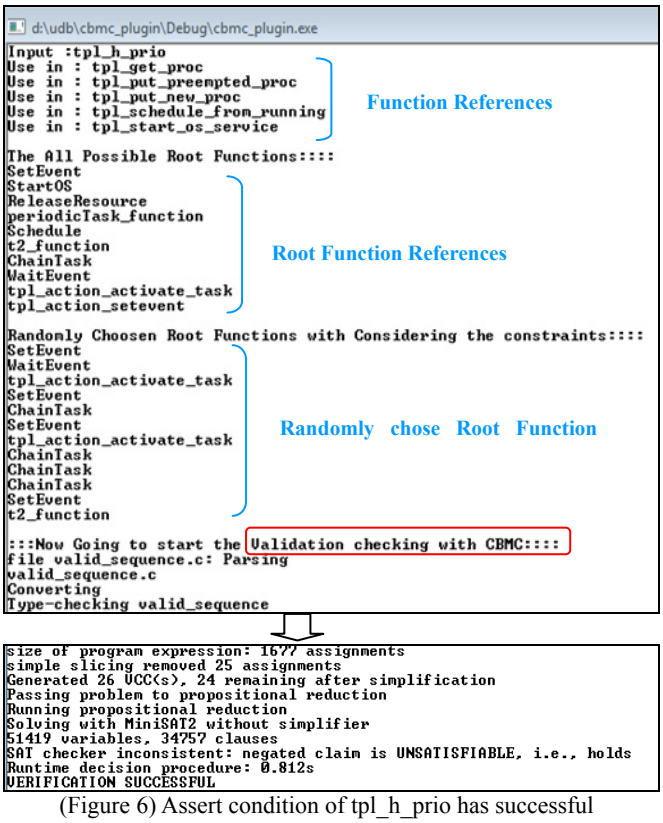
5. Experiments

This section briefly describes on the verification result using CBMC and the generated scenarios. From Trampoline OS we have chosen six variables used in assert conditions, which are tpl_h_prio, tpl_fifo_rw, tpl_ready_list, tpl_kern, prio, tpl_locking_depth (Table 2). For tpl_h_prio the procedure is represented below with details information.

The target variable is, tpl_h_prio, is used in assert condition 1 and 2 in Table 2. One of the scenarios for tpl_h_prio is represented in figure 6, which results in a successful verification by CBMC. Table 3 is presented the run time properties (Number of generated verification condition, Size of program expression and the runtime) based on different assert conditions.

```
1. main(){
2.   Take a Input of a Global variable, used in assert condition;
3.   for(i=1 to number of function references)
4.     func_ref[i]=name of each function reference(i);
5.   for(i=1 to number of function references){
6.     for(j=1 to number of root function references){
7.       if(name of root function reference has not stored before){
8.         root_ref[j]=name of each root function reference(j);
9.         for(k=1 to number of calls functions from the root_ref[j]){
10.          call_func_name= name of calls function(k);
11.          for(L=1 to number of function references){
12.            if(call_func_name==func_ref[L]){
13.              /*n initial value 0*/
14.              scenario[n][0]=root_ref[j];
15.              scenario[n][1]=func_ref[L];
16.              n++;
17.            } //if
18.          }//for L
19.        }//for k
20.      }//if
21.    }//for j
22.  }//for i
23.  for(i=1 to 100){
24.    rand_name= randomly chose an root function's name;
25.    for(j=1 to n){
26.      if(scenario[n][0]==rand_name){
27.        constraint_check(rand_name);
28.        if(constraint checking successful)
29.          record the call sequence of function;
30.      }//if
31.    }//for k
32.  }//for I
33.  Verify the whole scenario using CBMC;
34.}//main
```

(Figure 5) Pseudo code of the process



(Figure 6) Assert condition of tpl_h_prio has successful

Assert Condition	Remarks
1. assert (tpl_h_prio != -1);	Verification Successful
2. assert((tpl_h_prio >= 0) && (tpl_h_prio <3));	Verification Successful
3. assert(tpl_fifo_rw[tpl_h_prio].size> 0);	Verification Successful
4. assert(tpl_fifo_rw[prio].size<tpl_ready_list[prio].size);	Verification Successful
5. assert(tpl_fifo_rw[prio].size==(size_before+1));	Verification Successful
6. assert(tpl_kern.running!= NULL);	Unable to derive because of tpl_kern deal with pointer
7. assert((tpl_kern.running->state)==RUNNING);	Unable to derive because of tpl_kern deal with pointer
8. assert((prio>=0)&&(prio<3));	Verification Successful
9. assert(tpl_locking_depth>=0);	Verification Successful

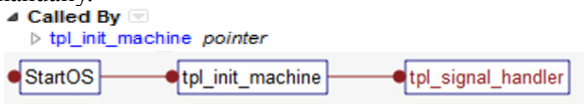
(Table 2) Assert conditions in Trampoline OS

Assert No.	Num. of Generated VCC(s) (verification cond.)	Size of program expression(assignments)	Runtime
Assert Cond. 1	12	1663	0.758s
Assert Cond. 2	22	1673	0.809s
Assert Cond. 3	10	659	0.555s
Assert Cond. 4	6	343	0.625s
Assert Cond. 5	14	663	0.545s
Assert Cond. 8	3	181	0.407s
Assert Cond. 9	9	309	-

(Table 3) Run time data in verification time

6. Limitations

There are some limitations in this approach. If a function is a function pointer type then understand tool incapable to traverse onward. Figure 9 shows that *tpl_signal_handler* function is called by *tpl_init_machine* and *tpl_init_machine* called by *StartOS* (Figure 7). But because *tpl_init_machine* is a function pointer Understand cannot traverse *StartOS* node. Such cases are fixed up to manually.



(Figure 7) Called-by Graph of tpl_signal_handler using Understand API

We also need to consider “Unknown” data type variable, which mean the Understand analysis tool was unable to determine the type of the variable. It can be global variable or local variable. If it’s a local variable it will have no effect with each other, but for a global variable, the modifying value will have effect to others. So with that possibility we consider this for verification.

Initially to connect with the Understand database it takes approximately 2.3sec delay. So, it is unclear at this point whether our tool would scale well as the size of the database becomes larger. We may need to consider the performance improvement.

7. Future Work and Conclusion

We have represented a method for automated valid scenario generation and verification of trampoline OS. The important key facts are: (a) Scenarios are generated referring to the function call sequence (calls 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:

- The limitation in pointer issue can be solved by tracing the pointer type manually or by using other analysis tools with the capability to deal with pointers
- 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.

Reference

[1] Clarke, E., Kroening, D., Lerda, F.: 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, pp. 168–176. Springer, Berlin (2004)

[2] Latella, D. and Massink, M. On testing and comformance relations for UML statechart diagram behaviors. Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis, Roma, Italy, July 22–24, 2002. ACM Softw. Eng. Notes, 27, 144–153.

[3] Sapna P.G. and Hrushikesha Mohanty. Automated Scenario Generation based on UML Activity Diagrams. IEEE, 2008.

[4] 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, pp. 437, 2004.

[5] Guoqing Yang, Minde Zhao, Lei Wang, Zhaohui 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.

[6] Understand Source code analysis and Metrics. http://scitools.com/index.php

[7] OSEK Group. OSEK/VDX Operating System Specification. http://www.osek-vdx.org.

[8] R.-T. S. group IRCCyN. Trampoline. http://trampoline.rts-software.org/, 2005.