A Function Network Analyzer for Efficient Analysis of Automotive Operating System

Lu ZhengYu ,Yunja Choi
Dept. Of Computer Science and Engineering Kyungpook National University
e-mail: luzhengyu1990@gmail.com

Abstract

This work developed a code analysis & extraction tool named Function Network Analyzer (FNA) to reduce the cost of software safety analysis. FNA analyzes functions and variables which a given functio-n depends on, and extracts subset of code that can be compiled of automotive operating system, final res-ulting a well-ordered code sequence that can be compiled for model checking technique. And the experi-mental result illustrates that FNA can get 100% accurate rate and over 96% reduction rate by testing AP-I functions from trampoline system.

1. Introduction

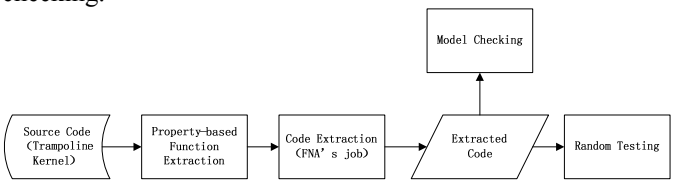
Automotive operating system that offers programmers the platform to develop automotive embedded software is safety-critical; it has a vital relationship with driver's life. Because any automotive embedded software operates on the top of such operating system and uses a part of its code. Therefore, we must perform comprehensive and rigorous analysis of the operating system. However, the comprehensive software analysis is very costly to achieve in practice.

To address such issue, this work developed a code analyzer and extractor named Function Network Analyzer(FNA) for automotive operating systems based on OSEK/VDX[3] international standard. FNA analyzes functions and variables which a given function depends on, and extracts a subset that can be compiled of the system code based on the analysis result. FNA is implemented by using a static analysis tool Understand[1] and MFC. The tool is applied to an open source automotive operating system Trampoline[4], shown a reduction of the target verification code 96.14% on average.

This paper is organized as follows: Section 2 states the related works, background of this work; Section 3 displays the design and implementation of FNA; Conclusion and future work of this project are presented in Section 4.

2. Background

Automotive operating system is the core part of automotive control software. Any careless mistake in the software would cause a tragedy related to person's life. Therefore testing and model checking [6] which is used for functional safety analysis are necessary for this kind software. And moreover, in the model checking technique, size of model/code to be verified needs to be minimized to avoid state-space explosion. Therefore, extracting code for the reduction of verification target and minimization of environment model's behavior from automotive operating system is part and parcel of model checking. However, comprehensive analysis of such operating system is a big cost and unpracticle by manual labour. So researching this domain is well worthy. The following diagram is a part of verification procedure of automotive software, showing the relationship of FNA, automotive operating system and model checking.



(Figure 1) Relationship between FNA and automotive verification procedure

Actually there already have some previous work in this domain. Reference[5] exploits the same subject, but it has several shortages which make its result cannot be compiled.
(1) Some functions cannot be found(e.g. tpl_shutdown).
(2) It cannot get Macro definition correctly(e.g. NULL).
(3) Some arrays' body cannot be found(e.g. tpl_ready_list, tpl_kern, and tpl_stat_proc_table).

For the reason that it mainly invokes Understand APIs <udbListReference> and <udbIsKind> to analyze a target function of automotive operating system, in this case, it cannot ensure that it could analyze code completely and comprehensively,when it encounts some special entities. It omits related entites that those entities depend on. Incomplete code makes its result cannot be compiled, its program is meaningless.

Among its problems, missing analysis of some arrays' body is the main one. What we need to do is to change the analyzing method with regard to those arrays. So we proposed a lexer method to fix these problems in follow sections.

3. Function Network Analyzer

FNA extracts all the related entities(one function, one variable or one typedef all can be an entity which is an umbrella name of different kind of definition of source code) of a target entity from a automotive operating system e.g.

* This work was partially supported by the National Research Foundation of Korea Grant funded by Korean Government (2012R1A1A4A01011788).

trampoline, and lists the content in a window. It also allows users to generate files containing the code it has extracted with the .c or .h file extension. Finally these files can be executable in the Linux OS. For more detailed explanation of FNA's functionality, the following example will illustrate

```
typedef  int  sig_t;
typedef long sig;
typedef unsigned char u8;
typedef unsigned int u32;
struct DEF
{
    u8 var1;
    u32 var2;
};
struct DEF1
{
    sig_t var3;
    sig var4;
};
void funcB()
{
    int a;
    typedef DEF1 def1;
}
void funcA()
{
    funcB();
    typedef DEF def;
    u8 var1;
}
```

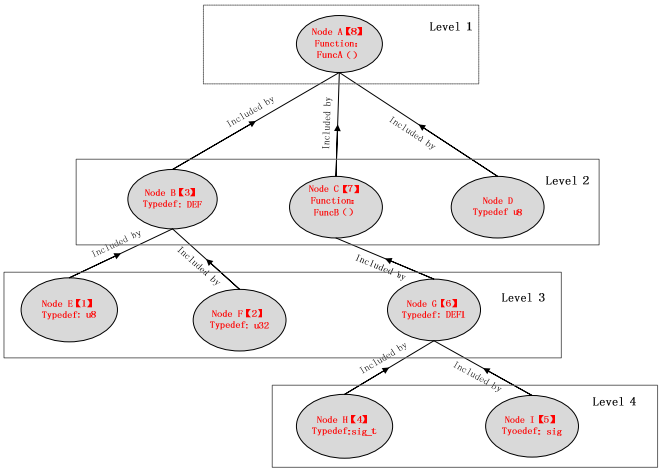
(Figure 2) Example to explain FNA

(1)Function funcA depends on: function funcB, struct DEF and typedef u8.
(2)Function funcB depends on: struct DEF1.
(3)Struct DEF1 depends on: typedef sig_t and typedef sig.
(4)Stuct DEF depends on: typedef u8 and typedef u32.

FNA analyzes the funcA. But funcA depends on other entites, so FNA continues to analyze these entits until there has no more new entity is depended. Then getting these entities content from source code.

The order of result needs to be considered as well when designing FNA. In case that typedef unsigned char u8'postion in (Figure 2) underneath function funcA' position, all the code cannot be compiled, So FNA not only should analyse a target entity completely, but also do the job of forming a well-ordered result that can be compiled.

3-1 The train of thought to design FNA



(Figure 3) Entity Tree for the code in Figure 1

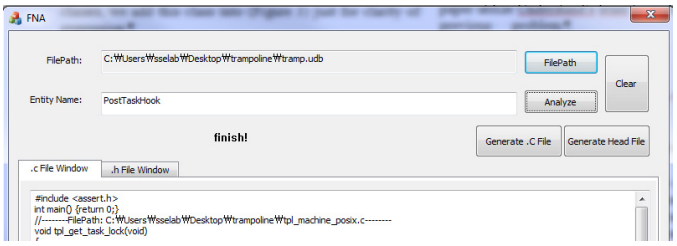
(Figure 3) displays a tree simulating an include-included relationship between entities after FNA analyzing a target entity function funcA. As we have discussed in(Figure 1). If we extract node A which has 3 offsprings, node B, node C and node D respectively. We also need to extract its 3 children, then get a list of children of A including B,C and D. And then analyzing the node B firstly, if it still has other children (and it really has). Then we extract its children, node E and F. Here we also get a list containing the node B's children. After that we analyze the node E firstly to check whether it also have other children. However, it is a typedef (typedef unsigned char u8) from which we cannot extract any user_defined identifier. So we stop here and get this node's body. The rest nodes can be done with the same manner. Finally the order of extracting node's body is : (1)E(2)F(3)B(4)H(5)I(6)G(7)C(8)A

Since FNA has already extracted the typedef u8's body in the node E. Therefore it ignores the node D. Till now the readers who have the algorithm knowledge background would recognize that this method of extracting the code is DFS(Depth-First-Search)

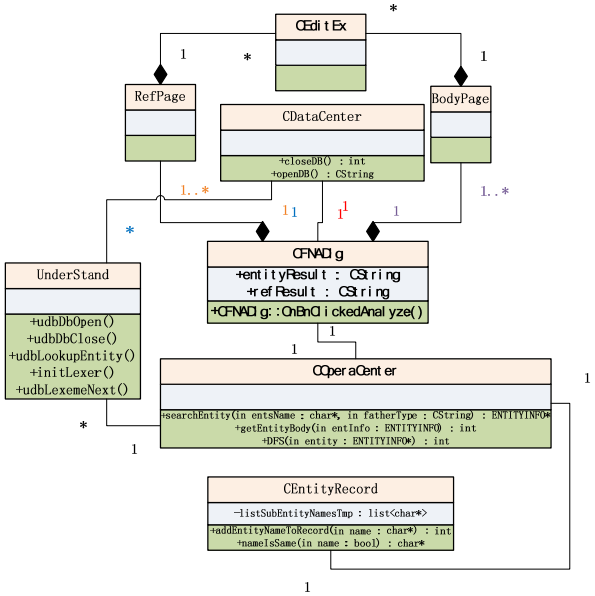
DFS always extracts entities's body from deepest level first, like(Figure 3). So the result is well-ordered. However we still confront an order problem even when we use the DFS in FNA. That will be discussed in following sections. The above content is the main idea when designing the FNA.

3-2. The detailed design and implementation

For the user-friendliness and operability, this program uses the MFC which can support graphic use interface. The following (Figure 4) displays the interface of FNA.



(Figure 4) Interface of FNA



(Figure 5) Class Diagram of FNA

Two edit controls which are FilePath and Entity Name respectively on the interface are used for user input. And two tab controls, .c File window and .h File window that display the result that should be put into a C file and a header file are used for outputting.

FNA includes 4 utility classes assisting its graphic interface, CEditEx, CFNADlg, CFileWindow, and HeadFileWindow. CFileWindow and HeadFileWindow are used for the implement of tab controls on FNA's interface. CEditEx offers the select-all and copy functionality for tab controls. Class CFNADlg which is generated by MFC automatically is applied to operate buttons on the interface(e.g. button analyze) . And moreover this class utilizes class CDataCenter which is listed in (Figure 5)

(Figure 5) shows a class diagram of FNA. CDataCenter is in charge of opening and closing a datasource whose file is end with udb extension. Class CEntityRecord is used for storing the information of entities that have been extracted

The class UnderStand, as a matter of fact, does not exist in FNA project, FNA only uses several important API functions of it having been listed in diagram in different classes, we add this class into (Figure 5) just for clarity of expression.

Class COperaCenter that contains the functions to extract code and generates final consequence is most crucial among all classes. Now we explain some important functins in the class CoperaCenter.

(1)searchEntity: When we found an entity having unknown type which may be a function or a local variable or a struct, it use the Understand API <udbLookupEntity> to check whether this entity is existing then to judge its type and save its information into a struct data structure, finally retun this struct's pointer.

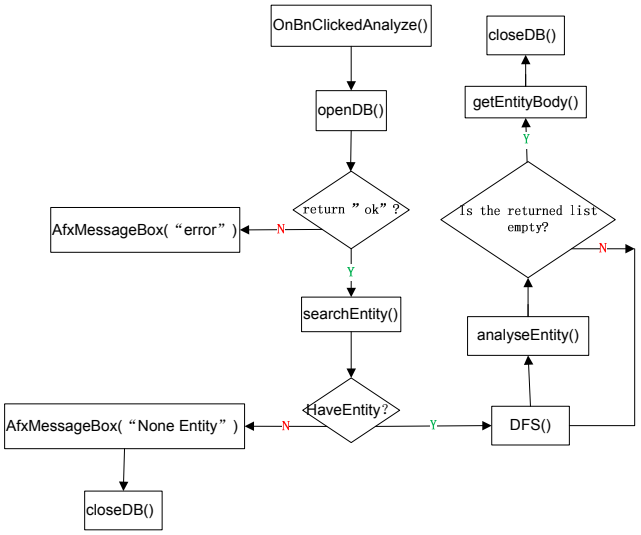
(2)analyseEntity: Analyzing functions and variables or other type of souce code which a given entity depends on,then put them into a list that will be returned at the end of this function.

(3) DFS function: It is mainly to implement the Depth-First-Search algorithm.

(4) getEntiybody: Putting a given entities'source code and its related entites's code into string for finally outputting.

There are also other functions in these classes, they all do some restrictive work making sure get the correct result, for example function nameIsSame can avoid that we extract the same entity by comparing the record so that can reduce the running time and get the unique result. Next step we connect these functions together

(Figure 6) illustrates the flow and connection of these main functions in (Figure 5). When a user clicks the button Analyze to analyze a target function, the FNA invokes function OnBnClickedAnalyze responding user's click action, then datasource is opened. And FNA begins to search this target entity. If this entity exists, FNA start to analyzes it using DFS algorithm until there has no entity it relies on. After that FNA get the entity one by one. Finally, FNA closes the datasouce.



(Figure 6) Program Structure Diagram

3-3. Comparison

Reference[5] analyzes a target entity by invoking Understand API < udbListReference >. In this way, it mainly fails to analyze arrays completely which include members in their bodies. For example the array in (Figure 7). It cannot analyze and extract this array's member. For the reason that, <udbListReference> cannot detect this array's body. comparing with that work, this paper utilizes Understand's lexer and lexem to analyze an entity to solve this problem. (Figure 8) shows the pseudo-code to implement the analyzing procedure by using lexer

```
tpl_kern_state tpl_kern =
{
    ((void *)0),
    &idle_task_static,
    ((void *)0),
    &idle_task,
    (2 +0),
    0x0
};
```

(Figure 7) code example

```
1 lexem = initLexer(entity location);
2 udbListReference(one target entity, &ref);
3//using reference filter to get target's begin and end position
4 int beginLine = udbListReferenceFilter(&ref);
5 int endLine = udbListReferenceFilter(&ref);
6 from the begin position;
7 while(lexem does not arrive the end of file)
8{
9  if (arrive the end position) break;
10 token = udbLexemeToken(lexem);
11   //token is a flag of an identifer defined by Understand
12   if (token == Udb_tokenKeyword)
13   /*
14   Udb_tokenKeyword is a marco defined by Understand
15   including the user-defined thing and system-definded
16   things in an entity
17   */
18   {
19      put it into a recod list
20   }
21   lexem = next lexem;
22}
```

(Figure 8) pseudo-code of analyzing entities by l-exer

If we use the lexer to analyze it, firstly we need identify the beginning and ending location of this array with the help of <udbListReference>. After initializing a lexer for this

array in its location, the lexer returns a lexem which is type of char* and that is the word “tpl_kern_state”.

Next checking this array’s lexem from “tpl_kern_state” to end ”}” by using a while loop. During this procedure, if it gets a token/keyword by invoking<udbLexemeToken>, it puts token into a record list. Finally all entities related to this array are in this list.

3-4. order problem of cross referencing problem

We have initiated this issue in the section 3-1, even using the DFS algorithm it also can meet the order problem in a very special condition, that is cross referencing problem.

<Table 1> cross reference example

Entity name	Source code of entity body
t1	typedef struct T1 t1;
T1	struct T1{ typedef struct T2 var1; char* var2;};
T2	struct T2{t1 var3;};

For example the entity t1 in <Table 1>, it depends on T1 and T2. If FNA analyzes it, final result order is:(1)T2(2)T1(3)t1. But we notice that T2 depends on the target entity t1. And t1’s location is below T2’s in the final result. So the result is not compiled in this order. This is the cross referencing problem. If we put the t1’definition before the rest code, it can work.

So we put all the one-line typedef result in the beginning of header file. To achieve this is simple. Because in this app it get an entity’s body by using the function FileRead which is called by function getEntitybody(in the figure 5)to read source file line by line. So first we can check a target entity to see whether its type is typedef then using FileRead to read one-line typedef. If we put all typedefs including mutil-line typedefs at the beginning of header file. It can cause order error when extracting function StartOS of trampoline. Therefore this irregular method is needed to be improved

3-5 Testing

The size of code extracted from automotive operating system need to be minimized for model checking. So reaching a higher reduction and guaranteeing the correctness of result are the main goals of testing. In this part we chose several system API functions of trampoline to test on this software and the software in reference[5].

<Table 2> Reduction of code size using FNA

Function name	Compiled	Extracted code's line num	Reduction percentage(Total line num 17888)
ActivateTask	yes	771	95.69%
tpl_put_new_proc	yes	771	95.69%
GetTaskID	yes	269	98.50%
ShutdownOS	yes	133	99.26%
GetEvent	yes	806	95.50%
ChainTask	yes	813	95.46%
TerminateTask	yes	771	95.69%
WaitEvent	yes	839	95.31%
CancelAlarm	yes	1035	94.22%

<Table 3> Problems of the existing tool[5]

Function name	Compiled	Problems	Extracted code's line num	Reduction percentage(Total line num 17888)
ActivateTask	no	entites in the specical arrays can not be found	520	97.10%
tpl_put_new_proc	no	find nothing	2	99.99%
GetTaskID	no	same as ActiveTask	242	98.65%
ShutdownOS	no	global object tpl_cpt_os_task_lock can no be found	128	99.29%
GetEvent	no	same as ActiveTask	250	98.61%
ChainTask	no	TaskRefType is not found	539	96.99%
TerminateTask	no	same as ActiveTask	467	97.39%
WaitEvent	no	same as ActiveTask	484	97.30%
CancelAlarm	no	tpl_proc_state* can not be found	228	98.73%

Comparing <Table 2> with <Table 3>, although existing tool reach a higher reduction of code, but its result is not correct and uncompilable. Although the result generated by FNA shows a 96.14% reduction on average which is little lower than the existing tool. But this result is able to be compiled. And this result has dramatically cut the size of trampoline kernel code. It is acceptable.

Actually these functions in tables are just a part of functions in trampoline. We have tested overall system API functions whose name begin with uppercase alphabet, they all can be compiled and the reduction rates are all above 90%. As for the running time of this app, generally one system API function takes around 10 seconds to get result and other simpler entity, for example structs and global objects takes around 5 seconds. So the efficiency of this program is kind of acceptable.

4. Conclusion and Furture work

This paper presented a new method lexer to detect and analyze entities more completely comparing with the work in reference[5]. And experimental result shows a 100% accurate rate and 96.14% reduction rate on average on the top of trampoline system. Finally we achieved a part of functionality assisting for the model checking verification technique. As for the future work, if trampoline system updatas version, we also should test FNA on those versions, and the unregular method that dealing with the cross referencing problem still needs to be improved. We wish this app can be a little helpful for the researchers who are working on the model checking of automotive embedded software safety

Reference

[1]Scitools .http://www.scitools.com
[2]RICHARD JOHNSONBAUGH and MARCUS SCHAEFER “ALGOTITHMS”
[3]OSEK/VDX Specification. http://www.osek-vdx.org.
[4]R.-T. S. group IRCCyN. Trampoline. http://trampoline.rts-software.org/
[5]Min-Gyu Park and Yunja Choi A Property-based Code Extractor for Formal Code Verification 한국정보처리학회 춘계학술발표대회 논문집 제 17 권 제 2 호 (2010. 11)
[6] Mingyu Park, Taejoon Byun and Yunja Choi Property-based Code Slicing for Efficient Verification of OSEK/VDX Operating Systems Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems, 2012