



UPPSALA  
UNIVERSITET

IT 14 022

Examensarbete 30 hp  
Juni 2014

# Automatic Detection of Unspecified Expression Evaluation in FreeRTOS Programs

---

Shahrzad Khodayari





UPPSALA  
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet  
UTH-enheten**

Besöksadress:  
Ångströmlaboratoriet  
Lägerhyddsvägen 1  
Hus 4, Plan 0

Postadress:  
Box 536  
751 21 Uppsala

Telefon:  
018 – 471 30 03

Telefax:  
018 – 471 30 00

Hemsida:  
<http://www.teknat.uu.se/student>

## Abstract

### **Automatic Detection of Unspecified Expression Evaluation in FreeRTOS Programs**

*Shahrzad Khodayari*

Embedded systems are widely used in most electrical devices. They are often complex and safety-critical. Therefore, their reliability is significantly important.

Among many techniques to verify a system, model checking models a system into temporal logic and can be used to assert a desired property on it. CBMC is a Bounded Model Checker for ANSI-C and C++ programs.

In this thesis, We extended the CBMC tool to check and automatically detect a C/C++ code containing a form of unspecified behaviors, like function calls with arguments that exhibit side effects which might be easily unnoticed by the programmers. In addition, the code can be configured properly to be used for ARM Cortex micro-controllers and FreeRTOS softwares.

Handledare: Philipp Rümmer  
Ämnesgranskare: Bengt Jonsson  
Examinator: Philipp Rümmer  
IT 14 022  
Tryckt av: Reprocentralen ITC



# Acknowledgements

This is a master thesis submitted in Embedded Systems to Department of Information Technology, Uppsala University, Uppsala, Sweden.

I would like to express my deepest gratitude to my supervisor Philipp Rümmer, Programme Director for Master's programme in Embedded System at Uppsala University, for his patience in supporting continuously and generously guiding me with this project.

I would like to appreciate professor Bengt Jonsson for reviewing my master thesis and offering valuable suggestions and comments.

I would like to thank professor Daniel Kroening for helping me and providing updates of CBMC.

Sincere thanks to my husband and my incredible parents who gave me courage and support throughout the project.

# Contents

<b>1 Introduction.....</b>	<b>1</b>
Contributions.....	3
Structure of the thesis report.....	3
<b>2 Background.....</b>	<b>5</b>
2.1 Verification.....	5
2.1.1 Model Checking.....	5
2.1.2 Theorem proving.....	6
2.2 CBMC.....	6
2.3 FreeRTOS operating system.....	8
2.4 Sequence points.....	8
<b>3 Unspecified side effects.....</b>	<b>11</b>
3.1 Arguments Evaluation Order .....	11
3.2 A formal semantics for arguments evaluation order.....	13
<b>4 Algorithm of evaluation order side effect.....</b>	<b>17</b>
<b>5 Preparation of analysis .....</b>	<b>23</b>
5.1 FreeRTOS.....	23
<b>6 Implementation.....</b>	<b>25</b>
6.1 CBMC .....	25
6.2 CBMC extension.....	26
<b>7 Evaluation and case study.....</b>	<b>31</b>
<b>8 Related work.....</b>	<b>33</b>
<b>9 Conclusion.....</b>	<b>35</b>
<b>10 Bibliography.....</b>	<b>37</b>

# List of Figures

2.1	The model-checking approach	6
3.1	Grammar G	14
3.2	Grammar rules	15
4.1	Operators type in a function's arguments list	17
4.2	Evaluation order side effect algorithm	18
4.3	Expression Tree	19
4.4	Making Operators relation graphs for expression $(a+b)  b++$	20
4.5	Making Operators relation graphs for expression $++c[i]-c[j]$	21
4.6	Operators relation graphs	22
6.1	Data type of a node in ORG graph	28
6.2	Array indexes assertion	28
6.3	Enabling arguments checker	29
7.1	GCC 4.8.2	31
7.2	Coverity 7.0	32
7.3	Modified CBMC	32

# Chapter 1

## Introduction

The majority of computer devices are embedded systems. These days cellphones, cameras, home appliances, robots, industrial machines, traffic lights, trains, airplanes, and many other devices mainly contain an integration of computer systems. Embedded systems are often complex and safety-critical. As both their hardware and software complexity are significantly increasing, reliability moves into the center of attention and needs to be tested properly.

Testing could be done in different stages, while producing software and it is sometimes as complex and time consuming as developing the software. Therefore, it is more beneficial to find bugs at an early stage in software development and provide valuable feedback for developers, in order to find and fix such problems prior to building up the next modules or even next release.

Sometimes bugs are due to a bad usage of a documented library or API, because of not reading the whole manuals or misunderstanding them. An other case could be not only a programmer's mistakes but also by reason of using complicated programming languages, like C/C++. In fact, while C/C++ are the most widely used languages for developing such systems, they are counted as highly prone to errors. These errors might lead to very serious consequences, including unpredictable and inconsistent program behaviors, run-time errors and even system crashes. Consequently effective detection of such errors is necessary.

Many embedded system applications use a special operating system called Real Time Operating System (RTOS). FreeRTOS is an open source RTOS that is used for embedded platforms such as ARM, Cortex-M3, AVR and STM32. It is written mostly in C and offers a small and simple real time operating system. It provides one solution for many different architectures and development tools and it is known to be reliable. In thesis, we use FreeRTOS as a processor of targeting program which might contain unspecified behavior. For this purpose, we prepared a minimalist or simplified model of the FreeRTOS API in form of a C library.

For achieving error detection goal, there are variable verification techniques. Using formal methods are well-known, which mathematically specify and verify these systems. They give us a proper understanding of a system and reveal inconsistencies, ambiguities, and incompleteness that might otherwise go undetected[1].



One of the most widely used formal methods is model checking, a technique that relies on building a finite model of a system and checking if a desired property holds in that model. *Bounded Model Checking* (BMC) techniques are able to efficiently and statically detect the possibility of run time exceptions in low-level imperative code, i.e., due to erroneous use of pointers, arithmetic overflows, or incorrect use of APIs.

One of the most successful tools for automatic verification that implements the bounded model checking (BMC) technique, is the C Bounded Model Checker (CBMC) used for ANSI-C/C++ programs. There is a class of defects that is detected by this CBMC, while many other verification tools have not unnoticed yet. For instance, among many features, we emphasize more on its ability to check array bounds even with dynamic size, pointer safety during conversion of pointers from and to integers and user-specified assertions; moreover it models integer arithmetic accurately, and is able to reason about machine-level artifacts such as integer overflow. In CBMC, any sort of checking appears as a specification that comes to a boolean formula, which is then checked for satisfiability by using an efficient SAT procedure. As a result, either a counterexample is extracted from the output of the SAT procedure, in the case that the formula is satisfiable, or if the formula is not satisfiable, the program can be unwound more to determine if a longer counterexample exists[2].

We extended the *CBMC* tool to check and automatically detect C/C++ code containing one form of unspecified behavior in the C/C++ standard which might go easily unnoticed by the programmers. According to the C/C++ standard, the order in which the arguments to a function are evaluated is unspecified and it depends on many factors like the argument type, the architecture, the platform and the compiler. The standard dictates that a C/C++ implementation may choose the order in which the function arguments are evaluated. It is the programmer's task to take care of them and make sure that the program does not depend on the order of evaluation. However, there is a warning flag in C/C++ compiler like GNU, *-Wsequence-point*, which warns about code that may have unspecified semantics because of violations of sequence point rules in the C and C++ standards. However, the current approach is suboptimal and many complicated cases are not diagnosed by this option. For instance, through this flag, the side effect among the array indexes are not evaluated precisely. If two indexes are expressions that might get same value at some point in the code, the flag is not able to detect this case.

Eventually, we provide capability for our CBMC extension to be run on applications written in FreeRTOS. We added an option to the CBMC front-end to verify if a given C/C++ code contains no such kind of side effects in arguments of each function and warn the programmer if there is any evaluation order dependency in the code. More details about how it works will be described in later chapters.

# Contributions

This thesis presents a study to develop a method and an automated tool for automatic detection of software defects. The target programs are written using the FreeRTOS real-time operating system, compiled by the ARM Micro-controller Development Kit (MDK-ARM) and executed on ARM Cortex micro-controllers.

The starting point of this work was the existing bounded model checker CBMC. We extended *CBMC* to be able to model check C code for ARM Cortex micro-controllers and automatically detects software defects in FreeRTOS softwares such as general C faults like function calls with arguments that exhibit side effects. It covers any kind of expressions containing variables, structures, classes, arrays and arithmetic operators over them. Moreover, we consider sequence points which force the compiler to evaluate the expressions in predefined order such as `||`, `&&`, `?:` and comma.

For evaluating the result, we compared our modified CBMC with the original CBMC and Coverity verification tool[15], [16] and GNU Compiler Collection (GCC) using `-Wsequence-point` flag. The result shows that the extended CBMC has the ability to check more unspecified behavior than the other three tools.

## Structure of the thesis report

In the second chapter, we will review some well-known verification techniques, CBMC tool, FreeRTOS platform and the concept of sequence points. Readers who are already familiar with these concepts could skip reading Chapter 2.

Chapter three describes and formally models one of the unspecified behaviors in C/C++ languages and talks about how it could exhibit side effects.

Chapter four introduces an algorithm for detecting these unspecified behaviors.

In Chapter five, we prepare a minimalist model of the FreeRTOS API in the form of a C library, in order to support detection of defects that might happen in FreeRTOS software specific to the MDK-ARM compiler and we explain why it is essential.

Chapter six briefly covers how we developed the front-end of CBMC tool to process a program with a set of software defects.

In Chapter seven, we will evaluate our tools with a relevant case study and we close with results of model checking.

Chapter eight discusses about other related works and tools in this area.

Conclusions and possible future work are presented in the last chapter.



# Chapter 2

## Background

In this chapter, we briefly introduce important concepts and tools that are used in the thesis. In the first section, verification terminology and its methods are described. The second section introduces the CBMC tool and its significant features. In the third section, FreeRTOS is referenced as an operating system environment, which is used in many embedded system applications. The last section talks about the sequence points for evaluating the expressions in C/C++ language.

### 2.1 Verification

Verification is a procedure of evaluating if a system meets a specification or imposed requirements. To verify a system, among many formal methods, model-checking and theorem proving are well-known. They are mainly used to analyze the system based on its specification for certain properties.

#### 2.1.1 Model Checking

Model checking requires building a finite model of a system, It checks whether a desired property holds in that model. There are several ways to model check C code such as Bounded Model Checking (BMC), model checking with predicate abstraction using a theorem prover, model checking with predicate abstraction using a SAT solver and translation of the C code into a model of an existing standard model checker [3]. The common property in all these techniques is an abstracted program with a finite state space that is gained from transformation of the system. Finiteness is required because the model checking algorithm should go through all states.

Bounded Model Checking, as the name suggests, does this transformation by unwinding possibly infinite constructs a finite number of times, for example, it executes *while* loops  $n$  times, where  $n$  is a limiting upper bound. A tool that implements bounded C model checking is *CBMC*. It is able to find a suitable  $n$  in most cases. However, if it does not succeed, there is a possibility for users to provide their own upper bound to be used by *CBMC*. In such a case, *CBMC* cannot guarantee that the user-provided upper bound is long enough to not miss any errors and that no longer counter-example is available. This is the case, where *CBMC* can only find errors and not prove correctness [3].

The advantage of model checking over theorem proving is that model checking can be used to check if a system is completely specified or to verify modules or partial specifications. It is completely automatic and fast and contributes useful information of system's correctness. The model checker will either terminate with answer true indicating that the model satisfies the specification or give a counterexample execution that shows why the specification is not satisfied, which can be useful while debugging (Figure 2.1) [4].

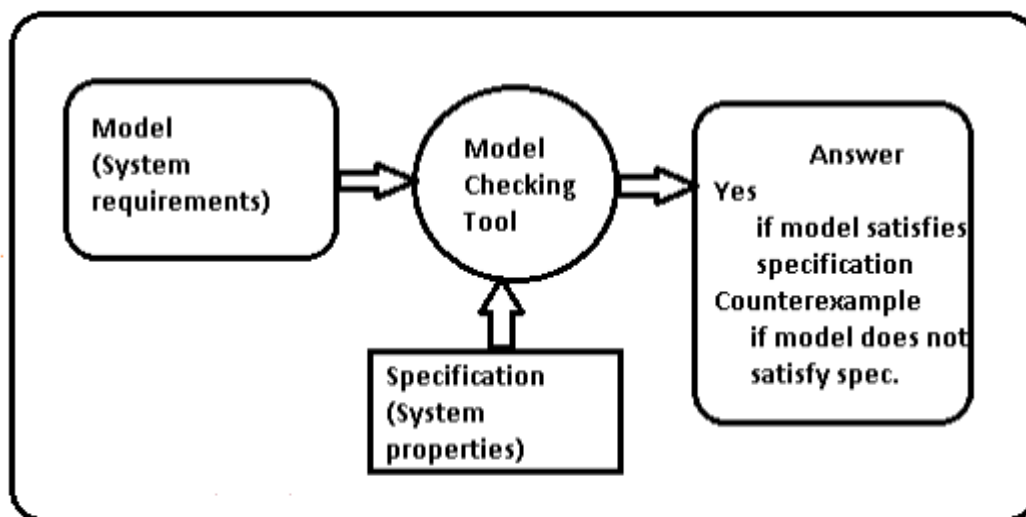


Figure 2.1: The model-checking approach

### 2.1.2 Theorem proving

In theorem proving, a system and all desired properties are revealed mathematically in formulas. This is given by a formal system, which defines a set of axioms and a set of inference rules. Then all properties that should be held by the system are being proved from the axioms of the system by applying the inference rules. It is essentially, a process of proving a property from the axioms of the system.

In contrast to model checking, theorem proving concerns infinite state spaces and proves these domains by structural induction techniques. Theorem proving mainly requires interaction with a human and humans might yield invaluable vision into the system and properties for being proved and it makes the process slow and sometimes error prone [1].

## 2.2 CBMC

CBMC is an open source model checker that uses bounded model checking technique to verify C or C++ programs. C/C++ files names are given to it as command line arguments. Similar to other compilers, it integrates all definitions and functions from each file but instead of making the binary code, it produces a goto-program of the program. The goto-programs are simplified C/C++ programs, which contain program's information such as variable's data type, any type casting and library functions, in a structured way and are represented in the form of Control Flow Graphs (CFG). In goto-programs, each variable is

assigned once and it is done by renaming in each case, this transformation is called Static Single Assignment (SSA).

In next step, a CNF is generated from this intermediate symbolic code and is passed to a SAT solver. SAT solver checks this equation's validness and it gives a counter-example trace when it fails. This shows that a bug is found in the program [2].

Considering the real time behavior of embedded systems, loop constructs are limited in number of iteration. CBMC verifies such finite upper run time bounds by unwinding all loops and checks if enough number of iterations are set in order to prove the absence of errors[2].

CBMC also provides set of keywords, which can be used to aide CBMC with more information about the program. These keywords can be used for program instrumentation. The program instrumentation is a procedure to verify some properties of the code. `__CPROVER_assert(expr)` and `__CPROVER_assume(expr)` macros are examples of these keywords.

The former can be used to check any condition (*expr*) with the same logic for assertions in the usual ANSI-C expression logic. When CBMC encounters this keyword, it tries to generate a formula to check assertion failure. The generated formula is verified using SAT-solvers. If the formula is satisfiable then assertion fails and CBMC generates error and produces counter-example showing possible trace of error. The latter macro, `__CPROVER_assume(expr)`, is used to restrict non-deterministic choices made by the program and it reduces the number of program traces that are considered and allows assume-guarantee reasoning [5].

CBMC also supports pointers, arrays, structures, floating point operations and function pointers.

There are other tools like BLAST [6] and Extended Static Checker for Java (ESC/Java)[7].

BLAST is a software model checker for C programs. Like CBMC, it checks that software satisfies behavioral properties of its interfaces and it uses counterexample-driven automatic abstraction refinement to construct an abstract model, which is model checked for safety properties. However, the advantage of CBMC over BLAST is that CBMC can also be used to verify consistency of hardware designs with a functional specification (written as C program). It can verify modules, and not only whole programs and it treats recursive functions and has GUI.

ESC/Java tool also attempts to find common run-time errors at compile time but in Java programs. It is based on simplify theorem prover using SAT checking and translates code to SSA, and then into verification conditions. ESC/Java supports assume-guarantee reasoning that are on methods and method calls, whereas in CBMC assume-guarantee statements can appear in any place in the program.

## 2.3 FreeRTOS operating system

Real time systems often run on special operating systems. A Real Time Operating System (RTOS) provides facilities to programmers such as process execution, predictability, data structures, and mechanisms for inter-process communication. FreeRTOS is used to develop real time systems for embedded devices.

FreeRTOS is designed to be small and simple. The kernel itself consists of few C files. To make the code readable, easy to port, and maintainable. It is written mostly in C, but there are a few assembly functions included where needed (mostly in architecture specific scheduler routines). FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores and software timers [8].

The fast execution, low overhead, configurable scheduler, co-routine supports, trace support and very small memory footprint are key features of FreeRTOS.

## 2.4 Sequence points

In C and C++ standards, the order of evaluating expressions is expressed by concept of sequence points. A sequence point shows which part of the expression is executed before and which one after it. Therefore, a partial ordering occurs between executions of different sides. For instance, sequence points could be after the first operand of operators `&&`, `||` and `?:`, in a function call after evaluation of its arguments but before executing the function body and in many other specific cases.

The common sequence points listed in the C++ standard are [9]

- at the end of the evaluation of full expression, (§1.9/16)
- in a function call (whether or not the function is inline), after the evaluation of all function arguments (if any) which takes place before execution of any expressions or statements in the function body, (§1.9/17)
- in the evaluation of each of the following expressions after the evaluation of the first expression, (§1.9/18)

$a \ \&\& \ b$	(§5.14)
$a \ /\! / \ b$	(§5.15)
$a \ ? \ b : c$	(§5.16)
$a , b$	(§5.18)*

\* Not when it is used as a separator between the arguments of a function. The behavior is unspecified in that case if  $a$  is considered to be a primitive type.

C and C++ standards set a rule that “Between the previous and next sequence point, an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.” [9]

which means that at each sequence point, the side effects of all previous expressions will be completed and it should be known when the effects of their operators occur. For example, in expression  $a \parallel a++$ , object  $a$  is modified and read at right and left sides of operator  $\parallel$  respectively and although the operator  $++$  has side effect, it doesn't have ambiguous effect in evaluating of expressions  $a$  or  $a++$ ; mainly because they occur in different sequence points (one from beginning to operator  $\parallel$  and the other afterward to the end). But if  $a$  and  $a++$  are in between two sequence points, as an example  $a + a++$  (operator  $+$  is not a sequence point), the rule is disobeyed and therefore evaluation of expression is not specified according to the standard.

However, breaking this rule is not too rare and even could say it is common. Programmers easily forget the rule and plenty of such places are left unspecified. There are many examples of this cases which are not sensible for programmer and easily neglected. Codes like:

$$\begin{aligned} &a + a++ \\ &a[i] + a[i++] \\ &a[i++] = i \end{aligned}$$

might be used frequently but are not reliable because there is no sequence point specified for the increment, post-increment, indexes and assignment operators. Moreover, sometimes these side effects could be found in the order in which the arguments to a function are evaluated.

In GNU, there are options to enable different warnings. Argument *-Wall* enable all warning about constructions that some users generally wish to check for and consider them questionable and easy to avoid (or modifiable to prevent the warning).

*-Wall* also turns on the *-Wsequence-point* warning flag which warns about code that may have unspecified semantics because of a sequence point rule violation in the C/C++ standard. However, it is not able to cover all possibilities and many complicated cases are not diagnosed by this option, in Chapter 3.1, we will see them in details.

Precise meaning of the sequence point rules is not explained clearly in the standard and it causes some debate over the rules. Alternatively, what was described as sequence point rules makes “only partial orders” such as the case when two functions are called within one expression with no sequence point between them; the order in which the functions are called is not specified. However, the standards committee has ruled that function calls do not overlap [10].





# Chapter 3

## Unspecified side effects

In this chapter, we describe how evaluation of function arguments could be seen as one of the common defects in C/C++. We express both expressions evaluation and defects in formal semantic. In the last part of this chapter, we explain our algorithm and show how these side effects in function arguments are detected.

### 3.1 Arguments Evaluation Order

As it mentioned earlier, bugs could occur due to bad usage of the documented rules of programming languages. This is quite common in C/C++ programs. Sometimes programmers forget to check if their codes are specified by the standard. More specially if the code has portable behavior and they can count on it. Our focus is on how this could be issued in evaluation of function arguments.

In section 1.9 of the C++ standard, all possible undefined, unspecified and implementation-defined behaviors were expressed clearly. Here we briefly explain these terminologies.

#### **Undefined behavior:**

Behavior, due to use of a non-portable, erroneous data or program construct, where the standard imposes no requirements for them. An example of undefined behavior is the behavior on integer overflow.

#### **Implementation-defined behavior:**

Behavior, where each implementation documents how the choice is made and the language provides a documentation describing its characteristics and behavior. An example of implementation-defined behavior is size of integer where the implementation must have only one definition for every place in the program.

#### **Unspecified behavior:**

Behavior, where a set of allowable possibilities is defined but it is not deterministic. The standard enforces no further requirements and the implementation is not required to document which option is chosen in any occurrence. For example, the compiler can choose different possibilities in different places, where the cases could even happen in the same

program.

Moreover, from the C standard specification, we mark the following cases that are not specified in the language [11]:

- Use of an unspecified value, or other behavior where the International Standard provides two or more possibilities and imposes no further requirements on what is chosen in any instance. An example of unspecified behavior is the order in which the arguments to a function are evaluated (§3.4.4)
- The order in which sub-expressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, `&&`, `||`, `?:`, and comma operators (§6.5).
- The order in which the function designator, arguments, and sub-expressions within the arguments are evaluated in a function call (§6.5.2.2)

According to the C++ standard [9], the order in which the arguments to a function are evaluated is given as an example of unspecified behavior. In fact, it depends on many factors like the argument type, the called function's calling convention, the architecture and the compiler. On an x86, the Pascal evaluates arguments left to right, whereas in the C/C++ calling convention it is right to left. Therefore, programs, which run on multiple platforms should take the calling conventions into account to skip any surprises, side effects or crashes. The standard dictates that a C/C++ implementation may choose in which order, function arguments are evaluated. To be in the safe side, the program itself should not depend on the order of evaluation of side effects and shall not use parameters of a function in default argument expressions, even if they are not evaluated.

By the following examples, we intend to clarify this common unspecified case according to the standard. Consider the function *test*:

```
void test(int arg1, int arg2, ...);
```

Assume that somewhere in the program there is a call like:

```
int i = 0;  
test(i++, i, ...);
```

How or in which particular order, different environments evaluate the arguments, is so important that even this simple function call can behave differently from one to other. For instance, *test(1, 1, ...)*, *test(1, 0, ...)* or even *test(0, 0, ...)* yeild possible results.

The second case is when arrays are involved; the index expressions come to center of attention.

```
int a[2] = {0, 1};  
int i = 0;  
test(a[i]++, a[i], ...);
```

But more interesting example is when we have different indexes of an array:

```
int a[2] = {0, 1};
int i = 0;
int j = 0;
test(a[i] ++, a[j], ...);
```

In this case, from the syntax point of view,  $a[i]$  is not  $a[j]$ . However, they might point to the same location of memory when  $i$  and  $j$  hold same value.

In addition, next example shows that the sequence point rule could effect these unspecified cases:

```
int i = 0;
test(..., i++ || i, ...);
```

The `||` operator is a sequence point and forces the compiler to evaluate its left and right operands in a specified order; then there is no unspecified behavior in this example.

Therefore, it may be necessary to warn the user, if evaluation of arguments of any particular function lead to unspecified behavior due to expressions with possible side effects.

However, the original CBMC allows all side effect operators with their respective semantic. Moreover, regarding the ordering of evaluation, CBMC uses a fixed ordering of evaluation for all operators. It believes, while such architecture dependent behavior is still valid in ANSI-C programs, showing these cases are not desirable [5].

Furthermore, we saw these side effect warnings as a demand and added this option to CBMC front-end, to verify that a given C/C++ code contains no side effects in arguments of its functions. In the following section, we present a formalization of argument expression through precise description of the C/C++ language interface.

## 3.2 A formal semantics for arguments evaluation order

In this chapter, a formal semantics of expression evaluation is presented. The Structural Operational Semantics (SOS) is used in this project, which is a set of rules for giving a formal semantics of expression. It basically defines the behavior of a program in terms of behavior of its parts and provides a structural view on operational semantics; in my opinion this structure is easy to follow.

An SOS rule is in the form of:

$$\frac{\text{assumption}, \text{requirement}}{\text{conclusion}} \quad (\text{name})$$

where the *assumption* is a pre-condition of an expression before its evaluation and *requirement* shows under which domin this assumption is hold.

Grammar G shows the syntax of an expression, figure 3.1. Although this simplified grammar of expressions is not fully matched to C/C++ languages, it covers most main types of operators with clear syntax similarity to C/C++. For the sake of simplicity, the similar operators are skipped in this grammar but it is easily extendable without extra complexity.

```

int_expr ::= var_access /
            int_expr bin_opr int_expr /
            int_expr seqpoint_opr int_expr /
            n
var_access ::= int_var /
            int_var++ /
            array_access /
            array_access++
int_var ::= x
array_access ::= a[int_expr]
bin_opr ::= + | - | * | / | = | <
seqpoint_opr ::= || | &&

```

Figure 3.1: Grammar G

where  $n$  is a literal number,  $n \in \mathbb{Z}$ ,  $x$  is an integer variable,  $x \in Var_Z$  and  $a$  is an array variable,  $a \in Var_{ar}$ , in the expression  $int\_expr$ . In this grammar, each expression may contain integer/array variables, binary/sequence point operators or literal numbers. It can be either a simple integer/array variable, e.g  $x$  or  $a[int\_expr]$ , or an expression with side-effect operators such as post increment,  $++$ . In addition to binary operators, which are commonly used in an expression, there might be sequence point operators which are special binary operators that effect the order of evaluation.

We explain the semantics of evaluating a program by describing how expressions are evaluated, rules are executed, states (scopes) are changed. Variables and arrays are evaluated to integers and sequences of integers, respectively. The evaluation may yield a new state either as a simple expression or as an expression containing an array. In general, we say that expression  $e$  in state  $S$  evaluates to expression  $e'$  and state  $S'$  and write:

$$(S, e) \rightarrow (S', e')$$

where

$$S : Var_Z \rightarrow Z \\ Var_{ar} \rightarrow (Z \rightarrow Z)$$

is state for either integer variables or array variables. By continuing evaluation of all intermediate expressions, we have the final value for expression  $e$  as follow:

$$(S, e) \rightarrow^* (S_f, v)$$

Where  $S_f$  is the post state and  $v$  is an integer value.

Each possible rule in grammar G is expressed according to SOS rules in figure 3.2, where notation  $\circ$  is used for binary or sequence points operators, and notation  $|$  shows the new state by updating the specified memory.

1	$\frac{x \in Var}{(S, x) \rightarrow (S, S(x))}$	<b>Variable evaluation</b>
2	$\frac{a \in Var_{ar} , \quad n \in Z}{(S, a[n]) \rightarrow (S, S(a)(n))}$	
3	$\frac{(S, e) \rightarrow (S', e')}{(S, a[e]) \rightarrow (S', a[e'])}$	
4	$\frac{(S, e) \rightarrow (S', e')}{(S, a[e]++) \rightarrow (S', a[e']++)}$	
5	$\frac{(S, r) \rightarrow (S', r')}{(S, r \circ l) \rightarrow (S', r' \circ l)}$	<b>bin_opr / seqpoint_opr evaluation</b>
6	$\frac{(S, l) \rightarrow (S', l')}{(S, r \circ l) \rightarrow (S', r \circ l')}$	<b>bin_opr evaluation</b>
7	$\frac{z = r \circ l , \quad r, l \in Z}{(S, r \circ l) \rightarrow (S, z)}$	
8	$\frac{r \in Z , \quad r \neq 0}{(S, r \circ l) \rightarrow (S, r)}$	<b>seqpoint_opr (   operator) evaluation</b>
9	$\frac{r \in Z , \quad r = 0}{(S, r \circ l) \rightarrow (S, l)}$	
10	$\frac{r \in Z , \quad r \neq 0}{(S, r \circ l) \rightarrow (S, l)}$	<b>Seqpoint_op (&amp;&amp; operator) evaluation</b>
11	$\frac{r \in Z , \quad r = 0}{(S, r \circ l) \rightarrow (S, 0)}$	
12	$\frac{x \in Var_z}{(S, x++) \rightarrow (S(x S(x)+1), S(x))}$	<b>Memory update</b>
13	$\frac{n \in Z , \quad f = S(a)}{(S, a[n]++) \rightarrow (S(a f(n f(n)+1)), f(n))}$	

Figure 3.2: Grammar rules

Different evaluation orders of complex expressions could lead to different state and final values and the absence of such a case can be verified by comparing each pair of sub-expressions and check if they are independent. This condition is expressed as below:

$$\text{If } (S, e_1) \rightarrow^* (S_{f1}, v_1) \text{ and } (S_{f1}, e_2) \rightarrow^* (S_f, v_2) \text{ and} \\ (S, e_2) \rightarrow^* (S_{f2}, v'_2) \text{ and } (S_{f2}, e_1) \rightarrow^* (S'_f, v'_1)$$

imply:

$$S_f = S'_f \text{ and } v_1 = v'_1 \text{ and } v_2 = v'_2$$

then:

$$e_1 \text{ and } e_2 \text{ are independent.}$$

This means for any two sub-expressions  $e_1$  and  $e_2$ , if we first start by evaluating expression  $e_1$  in state  $S$  and reach its final value  $v_1$  in state  $S_{f1}$  and continue with evaluating  $e_2$  in new state  $S_{f1}$  to reach its final value  $v_2$  in state  $S_f$ , we will have the both expressions totally evaluated and  $(v_1, v_2, S_f)$  is final value  $e_1$ , final value  $e_2$  and post state, respectively. Alternatively, if we start by evaluating expression  $e_2$  in state  $S$  and reach its final value  $v'_2$  in state  $S_{f2}$  and continue with evaluating  $e_1$  new state  $S_{f2}$  to reach its final value  $v'_1$  in state  $S'_f$ , we will have both expressions totally evaluated with  $(v'_1, v'_2, S'_f)$  as final value  $e_1$ , final value  $e_2$  and post state, respectively. In order to show that these two options have similar effect on the result, we check if  $(v_1, v_2, S_f)$  and  $(v'_1, v'_2, S'_f)$  is equivalent. We could then claim that the final values and states of evaluation of expressions  $e_1$  and  $e_2$  do not depend on which order they are evaluated. Therefore, they are independent in this sense.

Using a simple expression  $a[i] \parallel (i * i++)$ , we show how these SOS rules work as follow. Either of sub-expressions,  $a[i]$  or  $(i * i++)$  can be as a first candidate but operator  $\parallel$  is a sequence point operator and according to rule 5 in figure 3.2, it forces a predefined order of evaluation on its operands. Therefore,  $a[i]$  is chosen first and then using rules 6,  $(i * i++)$  is selected. Expression  $a[i]$  is evaluated by rules 3, 1 and 2. Evaluation of expression  $(i * i++)$  can be either through rules 5, 1, 6, and 12 or rules 6, 12, 5 and 1. However, they reach different final states due to order of performing rules 1 and 12. In fact, while rule 12 is changing the memory location,  $i$ , rule 1 reads this location value. So the final state of expression  $i$  depends on final value of expression  $i++$ . Therefore, they are not independent!

# Chapter 4

## Algorithm of evaluation order side effect

There are several constraints on how to evaluate expressions in C/C++ language standard. As mentioned before, the most significant one is that “between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored” [ISO90, x6.3]. Violation of this constraint might result in unspecified behaviors. In this part, we present our algorithm for checking it and we explain how we determine the existence of the side effect in evaluation order of the arguments to any function.

The action of side effects happen by changing the memory. Therefore, it is important that while evaluating certain expression, any pairs of read and write over certain memory location are seen as a potential side effect. Principally, operators like assignments, increment or decrement are counted as write operators. For instance, in the following function’s arguments, there are a few read and write pairs.

Function	<i>test(a + b // b++, ++c[i]- c[j]);</i>									
Expressions	<i>a + b // b++</i>					<i>++c[i]- c[j]</i>				
Variables	<i>a</i>	<i>b</i>	<i>b</i>	<i>c</i>	<i>i</i>	<i>c</i>	<i>j</i>			
Operators	+		+		++		++	-	<i>index</i>	<i>index</i>
Operator types	read	read	write	write	read	read	read	read	read	read
	read	read	read							

Example 4.1: operators type in a function’s arguments list

In example 4.1, each expression in argument list contains different variables representing different memory locations. Whether these memory locations are changed or read should be considered, depending on what kind of operator is used on each variable.

In order to check if any stored values (memory locations) are modified at most once, we model all writing and reading the content of each memory location specified in arguments of a function, as one single graph. We call this graph model an Operators Relation Graph (ORG). Therefore, for each function, we will have a group/list of graphs.

Hence, ORG is built for each operand/variable in all expressions of a function argument list. Vertexes in this graph represent either read or write operators. Besides, an edge shows that its corresponding vertexes are in the same sequence point. By this technique, we could keep



track of all accesses to a variable in whole function arguments list.

Basically, we go through the arguments list from left to right and whenever we reach a write/read operator, we insert a new vertex to its corresponding operands ORG. If a *sequence point* is encountered, all those vertexes that are involved in this sequence point will be connected to each other by edges.

Thus, in an ORG, any pairs of (*read*, *write*) or (*write*, *write*) which are not connected to each other (no edge in between them) shows a dependence as their variables are accessed and modified more than once. This means, the order of evaluating them effects the result.

When these ambiguous behaviors are detected, either we make a warning by specifying the operand location in the code, or in the case that the side effect is a semantic violation, we added a counter example of it. Figure 4.2 illustrates evaluation order side effect algorithm.

---

### Algorithm 1 Detecting evaluation order side-effect

---

1. **for** any function *func* **do**
  2.     *argList* := {arguments of *func*};
  3.     *exprTree* := *expression\_tree*(*argList*);
  4.     **for all** *expr node* **in** *exprTree* **do**
  5.         *check\_node*(*expr*);
  6.     **end for**
  7. **end for**
- 
1. *check\_node*(*expr*) **do**
  2.     **for all** operands, *opr*, **in** *expr* **do**
  3.         **if** *opr*'s graph **notin** *graphList* **do**
  4.             Insert a graph *opr* → *graph* for *opr* into *graphList*
  5.         **end if**
  6.         Insert *opr*'s operator *opr* → *operator* as a node into *opr* → *graph*
  7.         **if** *opr* → *operator* is a new *sequence point* **do**
  8.             Connect *opr* → *operator* to other nodes in *opr* → *graph*
  9.         **end if**
  10.        **for all** nodes, *opr* → *node*, **in** *opr* → *graph* **do**
  11.            **if** *opr* → *operator* or *opr* → *node* is *write\_opr* **do**
  12.                **if** *opr* → *operator* and *opr* → *node* is not adjacent **do**
  13.                    Warning! side effect at location *opr*
  14.                **end if**
  - end if**
-

---

```

15.      end if
16.      // add assertion to check if the index expressions point to
17.      // same memory location, if so the counter-example is shown
18.      if opr is array and opr→index_expr isnot opr→node→expr do
19.          add assert( opr→index_expr == opr→node→expr) to code
20.      end if
21.  end for
22.  // check a child node or sub tree, recursively
23.  check_node(opr)
24.  end for
25. end check_node

```

---

Figure 4.2: Evaluation order side effect algorithm

In detection evaluation order side effect algorithm, we use an expression tree [11] which contains all expressions of a function's arguments list. We build this tree while parsing the code phase. Looking back to example 4.1, we could draw the expression tree as follow:

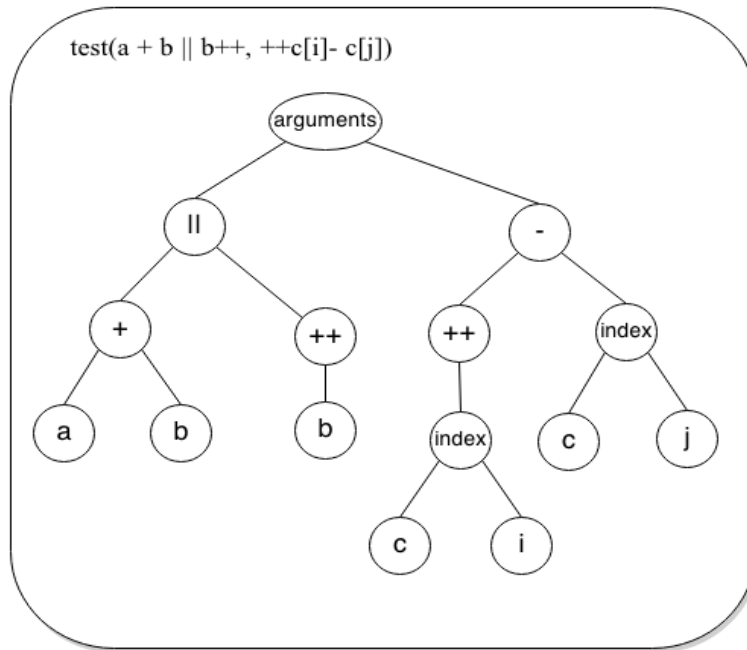
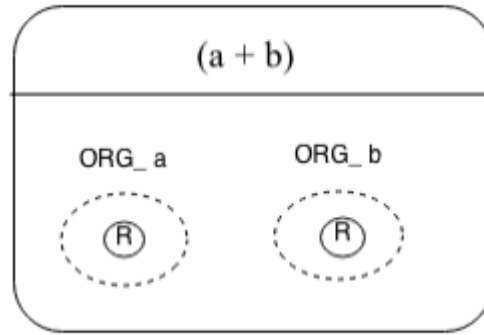


Figure 4.3: Expression Tree

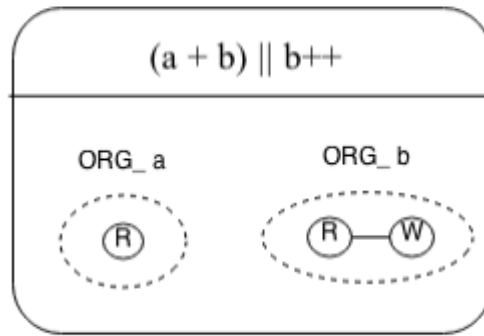
In this example, the algorithm traverses the tree in post order and makes ORG graphs for all variables. In general, when a new variable is met for the first time, one ORG graph is added to *test*'s graph list for that variable. Initially, the graph list is empty. The first node is *a*, which is met for the first time, so an ORG graph is assigned for it. After variable *a*, node *b* is the next seen, which is also a new variable, so another ORG will be added to the graph list. Next comes node *+*; as it is a *read* operator, a vertex should be added to each variable's

ORG involved in this operator, which means into  $a$ 's and  $b$ 's ORG graphs, Figure 4.4.a shows how the ORG appears to be at this point. Next nodes are  $b$  and  $++$ ; variable  $b$  is already seen and its ORG is available but  $++$  is a write type operator, so we add a write node to  $b$ 's ORG, figure 4.4.b.

When we reach to node  $||$ , we connect the read and write nodes in ORG<sub>b</sub> by an edge because operator  $||$  changes the sequence point. For ORG<sub>a</sub>, there is no change as variable  $a$  is only on one side of operator  $||$  rather than being on both the sides, figure 4.4.c. We should emphasize that edges are only between nodes from different sides of operator  $||$ .



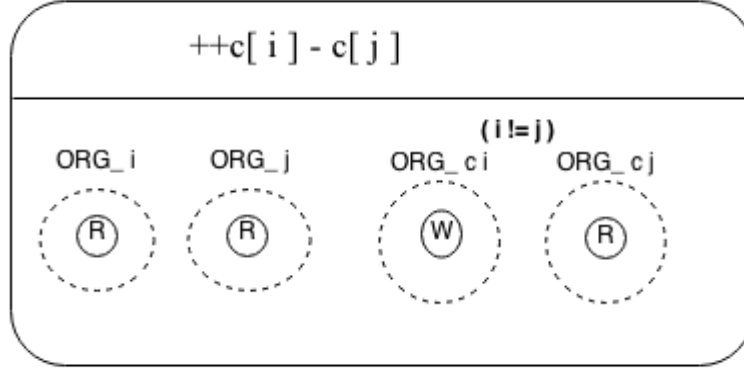
(a) ORGs for expression  $a+b$



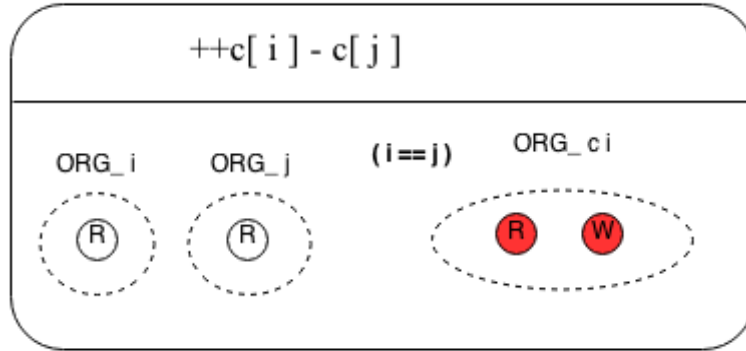
(b) ORGs for expressions  $a+b$  and  $b++$  (c) ORGs for expression  $a+b || b++$

Figure 4.4: Making Operators relation graphs for expression  $(a+b)||b++$

The second argument is  $(++c[i]-c[j])$ . Similarly, we add ORGs for variables  $c$ ,  $i$  and  $j$ . Figure 4.5.a illustrates the graph list for this expression. Besides, considering the array type, the algorithm also inserts an assertion to the code for the indexes expressions of  $c[i]$  and  $c[j]$  to check if they are equivalent; in this case  $assert(i==j)$ .



(a) ORGs for expression  $++c[i]-c[j]$  when  $i \neq j$



(b) ORGs for expression  $++c[i]-c[j]$  when  $i = j$

Figure 4.5: Making Operators relation graphs for expression  $++c[i]-c[j]$

Eventually, location  $a$ ,  $i$  and  $j$  are read only once in the arguments of *test* function. They do not depend on the order of evaluation.  $ORG_b$  has a pair of *read* and *write* which represents dependency between operators over variable  $b$ , but there is an edge in between, which shows those operators are in different sequence points. This implies that the order is predetermined. Therefore, no unspecified behavior happens in evaluation of variable  $b$ .

For array  $c$ , there are two possibilities. If  $i$  and  $j$  hold the different values, then two different locations of array  $c$  are involved. Such a case where  $c[i]$  and  $c[j]$ 's ORGs are illustrated as in figure 4.5.a shows no dependencies. However, if  $i$  and  $j$  hold the same value, this expression contains one memory location,  $c[i]$ , with one *write* and one *read* operator. Since there is no intervening sequence points between them, the expression result is independent to their evaluation's order. Consequently, according to algorithm 1, this case is reported as an unspecified behavior and the programmer will get a warning and a counter-example showing details, depicted by red nodes in figure 4.5.b.

Figure 4.6 shows all ORGs for this example.

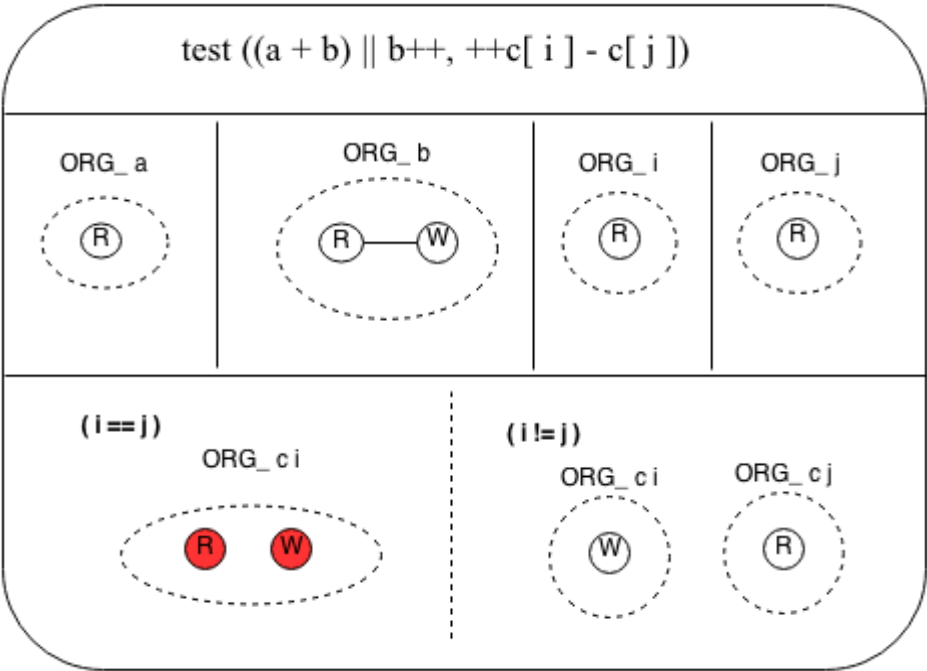


Figure 4.6: Operators relation graphs

# Chapter 5

## Preparation of analysis

### 5.1 FreeRTOS

In this thesis, we prepare the libraries of FreeRTOS and firmware of Cortex<sup>TM</sup>-M3 micro-controller as they are mostly used in many embedded projects. The chosen operating system and micro controller give the developer a range of facilities and they come with their own libraries, while the goal is to analyze the embedded project, these libraries come into attention, as well.

We choose Keil MCBSTM32C Evaluation Board platform for our implementation because of its popularity and a large number of STM32 devices that it provides. This board houses STM32F107VC with 256KB Flash, 64KB RAM, an ARM Cortex<sup>TM</sup>-M3 processor-based MCU running at 72MHz. Generally, the STM32 family of 32-bit Flash micro-controllers based on the ARM Cortex<sup>TM</sup>-M processor. The Cortex-M3 port includes all the standard FreeRTOS features like task priority assignment, Queues, Semaphores, Mutexes and etc. [12].

FreeRTOS is a real time kernel on top of which Cortex-M3 micro-controller applications can be built. It makes them to meet their hard real-time requirements that would result in absolute failure of the system. FreeRTOS organizes Cortex-M3 micro-controller applications as a collection of independently executing threads and decides which thread should be served first based on their priorities. How the priorities should be assigned to threads to ensure that hard real-time threads meet their processing deadlines is the application designer's task.

To be able to check any property by CBMC tool, specifically our interest, unspecified behavior of arguments evaluation order for functions of a code written in FreeRTOS and used the Cortex-M3 platform, we need to clarify all libraries for CBMC. In fact, many low level functions in the firmware and facilities in FreeRTOS are not typically considered in any test tools, neither in CBMC. Therefore, as first step, the minimalist version of FreeRTOS API is developed in the form of a C library. Basically, we replace both FreeRTOS and firmware files with our own versions that can later be processed by CBMC.

Some of the important files that have been modified in FreeRTOS and STM32F10x firmware are listed as follows:

## **FreeRTOS/**

*heap\_2.c*

The heap implementation but with some limitation. *heap\_1.c* and *heap\_3.c* are for alternative. Functions *pvPortMalloc()* and *vPortFree()* used to allocate and free dynamic memory blocks.

*list.c*

The list implementation used by the scheduler but it is also available for use by application code.

*port.c*

Portable layer API. Each function must be defined for each port for the ARM CM3 port

*queue.c*

Definition of the queue used by the scheduler.

*tasks.c*

Tasks creation, control, utilities API

*serial.c*

Basic interrupt driven serial port driver for UART0

## **Firmware/**

*stm32f10x\_gpio.c*

all the GPIO firmware functions

*stm32f10x\_rcc.c*

all the RCC firmware functions

*stm32f10x\_i2c.c*

all the I2C firmware functions

*stm32f10x\_exti.c*

all the EXTI firmware functions

*system\_stm32f10x\_cl.c*

ST STM32F10x Connectivity Line Device Series

*stm32\_eval\_ioe.c*

IO Expander driver for STMPE811 IO Expander devices

Supported features:

- IO Read/write : Set/Reset and Read (Polling/Interrupt)
- Joystick: config and Read (Polling/Interrupt)
- Touch Screen Features: Single point mode (Polling/Interrupt)
- TempSensor Feature: accuracy not determined (Polling)

*misc.c*

All the miscellaneous firmware functions (add-on to CMSIS functions)

*GLCD.c*

MCBSTM32C low level Graphic LCD (320x240 pixels) functions

Consequently, we are able to check for any defects in FreeRTOS application that are extended to CBMC. All the dependencies and configuration steps of porting CBMC to FreeRTOS project are done in a make file, which helps users to easily include this tool without making lots of settings, including all files paths and working with linking issues. This feature could be marked as easy configuration credit. In later chapters, we will examine all supported checkers by CBMC in the form of defects in a test project using the FreeRTOS and STM32F10x firmware.

# Chapter 6

## Implementation

This chapter shows briefly the modifications made to CBMC tool to be able to find possible unspecified behaviors in a given source program.

### 6.1 CBMC

The argument side effect checking, described in chapter 4, is implemented using C++ programming language. The source code is checked out from subversion (SVN) repository

<http://www.cprover.org/svn/cbmc/>

We used the *trunk* version for windows in this thesis. In order to reduce the amount of work required to set up a Visual Studio project for CBMC and the associated tools, a script is used which automates this process. The script is available in the CBMC SVN *trunk* in the directory "scripts" and is called "generate\_vcxproj". It could be configured by following command in a bash shell, e.g., provided by cygwin.

```
./generate_vcxproj
```

The command reads the Makefiles and automatically generates project files for cbmc, goto-cc and goto-instrument, and we can access them through Visual Studio. The project files come with filter definitions that order the source files according to the (top-level) sub directories they are in.

Note that the flex and bison tools and the irep\_id conversion tool still need to be run manually as mentioned in the compiling hint document.

This project file is helpful for debugging and building with MSBuild. For windows platform, CBMC still requires the pre-processor cl.exe, which is part of Visual Studio and the path to cl.exe must be part of the PATH environment variable of your system.

The *trunk* is structured in a similar fashion to a compiler. It contains language specific front-ends with limited syntactic analysis, intermediate format and a back-end tool for processing this format. Like a compiler, it takes the names of .c/.cpp files as command line arguments, then it translates the program and merges the function definitions from the various .c/.cpp files, just like a linker. But instead of producing a binary for execution, it performs symbolic



simulation of the program[13].

Here, we outline the trunk project but only the important directories with files that get modified, for the sake of clarity.

### ***/trunk***

#### ***/src***

All source codes are located in this directory and they are separated into different sub directories, such as, /analyses, /cbmc, /goto-programs, etc.

#### ***/goto-programs***

Contains the transformation program of the source code to an intermediate representation of C/C++ which is language independent. All converting methods are located here, and our new support is mostly added as a goto-program.

#### ***/cbmc***

The first full application is this directory. Here, the front ends (ansi-c, cpp, goto-program or others) are used to create a goto-program, goto-symex to unwind the loops the given number of times and produce an equation system. It then uses solvers to find a counter-example.

#### ***/goto-cc***

It is a compiler replacement that just converts C/C++ programs to goto-binaries. It is supposed to be dropped into an existing build procedure in place of the compiler. Thus, it emulates flags that would affect the semantics of the code produced. Which set of flags are emulated depends on the naming of the goto-cc/ binary. If it is called goto-cc then it emulates GCC flags, goto-armcc emulates the ARM compiler, goto-cl emulates VCC and goto-cw emulates the Code Warrior compiler. The output of this tool can then be used with cbmc[13].

#### ***/goto-instrument***

The *goto-instrument* is the top level control for the program. It could be used as a skeleton of new project. This directory contains a number of tools that are used in a goto-program. One can either modify it or perform some analysis. Here the command line is parsed to see which option is desired by user. It supports the following checks:

<code>--no-assertions</code>	ignores user assertions
<code>--bounds-check</code>	adds array bounds checks
<code>--div-by-zero-check</code>	adds division by zero checks
<code>--pointer-check</code>	adds pointer checks
<code>--arguments-check*</code>	adds argument order checks

\* not available in original CBMC

#### ***/analyses***

It makes a list of all checks that should be analyzed (e.g. options taken as

arguments by command line parsing).

*/doc*

The html and pdf versions of the source code documentation explaining the above directories more detailed [13].

We also need a SAT solver (in source). MiniSat2 is recommended by CBMC and it could be downloaded from:

<http://minisat.se/downloads/minisat-2.2.0.tar.gz>

## 6.2 CBMC extension

To design the argument-checker that was discussed in Chapter 4, we add a module to *goto\_programs* directory. Knowing some of the basic concepts might be useful here, such as, each function is a list of instructions, each of which has a type (one of 18 kinds of instructions), a code expression, a guard expression and potentially some targets for the next instruction. Our module checks each expression while it is being converted to an intermediate format referred to as *goto-binaries* or *goto-programs*. In conversion level, CBMC has a technique to adjust the code to standard definition in some special cases and prevent some side effects by cleaning expressions like `&& || ?`: comma (control-dependency), `++ --`, compound assignments, object constructors like arrays, string constants, structures and function calls. It actually rewrites the expression in a way that the standard specified. However, as we like to check expressions with more sensitivity, we need to do it before any cleaning to ensure nothing is missed or basically converted. In this regard, we make a list of identifiers of arguments list for each function. Each identifier represents a variable used in arguments in a function.

Figure 6.1 shows the data type for an ORG graph that is discussed in Chapter 4. Graph *cIdentifier* contains *m\_id* as unique ID, which is actually the variable's name and a list of all operators, *m\_operators*, which are involved with the corresponding variable in whole expressions of arguments list. Boolean *m\_bIndex* is used if the variable is an array type. Nodes in this graph are operators as mentioned earlier. Each node has a type *m\_opr* that shows whether the operator reads or writes the content of memory. When the operators of a variable are in the same sequence point, there is an edge between them in the variable's ORG graph. The edges between nodes are modeled by the *m\_seqPoint\_Id*, which contains the same value in two connected nodes. In the case that the variable has index as an array, *m\_arrayInx\_Id* is used for same purpose.

When the variable is an array, in order to be able to check if the content of same location is being considered, we keep track of its indexes with *m\_indexExpr*. Therefore, we could check if the operators are intending to manipulate and access the same location of an array more than once. For dynamic checking of this violation, we make an inequality expression from those index expressions, which we have in *m\_indexExpr*. And it will be inserted as an assertion node to expression tree in the same location in the goto- program, figure 6.2.

```

class cIdentifier
{
    std::string m_Id; //name
    std::vector<sOperator> m_operators;
    bool m_bIndex;
    ...
    // ACCESSORS and MANIPULATORS
    ...
};

```

```

struct sOperator
{
    oprType m_opr;
    std::string m_seqPoint_Id;
    std::string m_arrayInx_Id;
    index_exprt * m_indexExpr;
};

```

```

enum oprType
{
    READ = 0,
    WRITE,
};

```

Figure 6.1: Data type of a node in ORG graph

```

if(opr1->m_indexExpr == opr2->m_indexExpr)
{
    //expression of indexes
    exprt expr1 = opr1.m_indexExpr;
    exprt expr2 = opr2.m_indexExpr;

    //make an inequality expression of indexes
    exprt inequality(ID_notequal, bool_typed());
    inequality.copy_to_operands(expr1, expr2);

    //insert an assertion of the new expression to the code
    goto_programt::targett t = dest.add_instruction(ASSERT);
    t->guard.swap(inequality);
    t->location = expr1.location();
    t->location.set(ID_property, ID_assertion);
    t->location.set("user-provided", true);
}

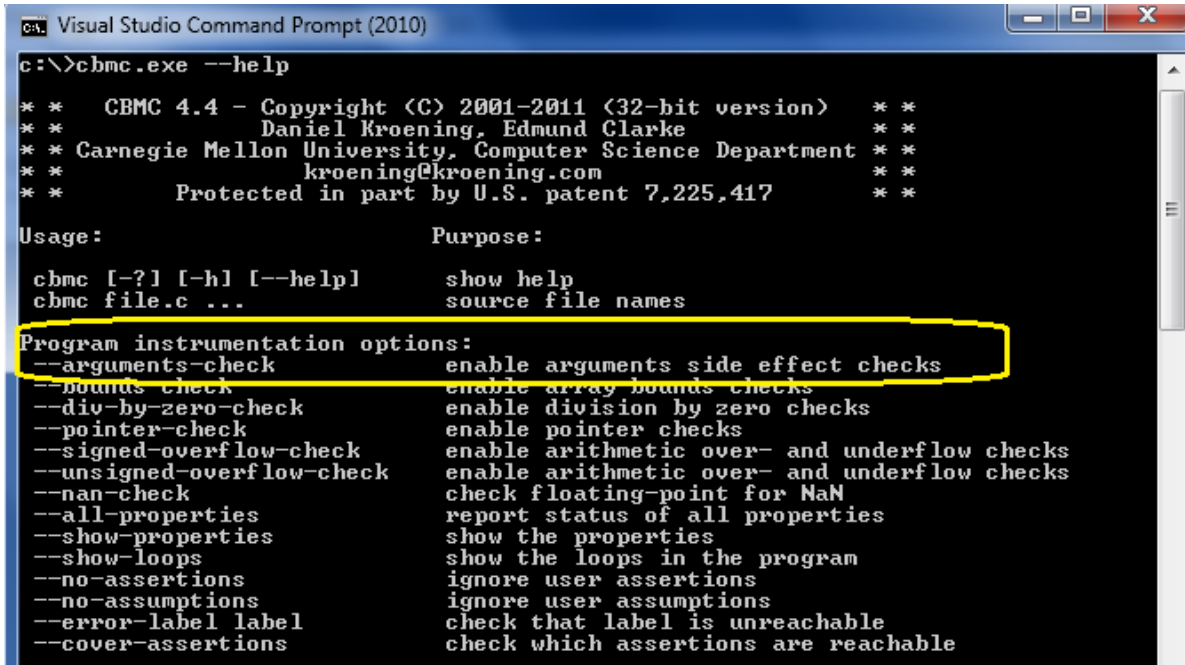
```

Figure 6.2: Array indexes assertion

Figure 6.2 shows a simplification of how an assertion is added automatically when two indexes of an array are involved in an expression. For any two operators *opr1* and *opr2* that effects the array indexes, it first checks if the expression trees of the indexes are equivalent. If so, then it prepares an assertion by making an inequality expression of indexes, *inequality*, and inserts it to the same line as the array indexes are located. This assertion has *user-provided* type which means it is not really inserted to the code but it is only available for further checking in CBMC solvers.

Parameter `--arguments-check` is added to program instrumentation options for instructing

CBMC to look for side effects while evaluating any function arguments. Through this configuration option, user is able to customize the behavior of CBMC as desired by enabling this checker. All parameters are disabled by default. CBMC `--help` gives the full list of the available options. Figure 6.3 shows that `--arguments-check` is also accessible in this list.



```
Visual Studio Command Prompt (2010)
c:\>cbmc.exe --help

* *   CBMC 4.4 - Copyright (C) 2001-2011 (32-bit version)   * *
* *   Daniel Kroening, Edmund Clarke                       * *
* *   Carnegie Mellon University, Computer Science Department * *
* *   kroening@kroening.com                                * *
* *   Protected in part by U.S. patent 7,225,417           * *

Usage:                                     Purpose:
cbmc [-?] [-h] [--help]                   show help
cbmc file.c ...                           source file names

Program instrumentation options:
--arguments-check                         enable arguments side effect checks
--bounds-check                           enable array bounds checks
--div-by-zero-check                       enable division by zero checks
--pointer-check                           enable pointer checks
--signed-overflow-check                   enable arithmetic over- and underflow checks
--unsigned-overflow-check                 enable arithmetic over- and underflow checks
--nan-check                               check floating-point for NaN
--all-properties                           report status of all properties
--show-properties                         show the properties
--show-loops                             show the loops in the program
--no-assertions                           ignore user assertions
--no-assumptions                         ignore user assumptions
--error-label label                       check that label is unreachable
--cover-assertions                       check which assertions are reachable
```

Figure 6.3: Enabling arguments checker



# Chapter 7

## Evaluation and case study

This chapter summarizes the result of model-checking performed some codes containing undefined behavior in their argument list of functions. We experiment the same code with desired dependency among a function's arguments through Coverity 7.0, GCC 4.8.2 and our modified CBMC.

The case code contains two types of dependency among arguments. For clarity, we inject them in separate functions.

---

```
13 int a = 0;
14 int c = a;
15 size_t order[3] = {1, 2, 3};
16 get_order(order[a], order[c]++);
17 get_order(order[a], order[a++]);
```

---

After testing the code by the mentioned tools, we observed that all three tools are able to detect some sort but not all kinds of evaluation order dependencies in both functions arguments. In this test code, line 17 is reported in all three compiling ways as evaluation order violation due to a pair of *read* and *write* operations over variable *a*. Similarly, we experienced more dependencies in variables of expressions with no array memories and all these tools found them successfully. However, in different type of dependencies the result was not the same; For example, in this code, in line 16, when indexes *a* and *c* of array *order* is read and written respectively, Coverity and GCC are not able to check whether these indexes hold same value and if the same location of memory is going to be processed or not. In contrast, our modified CBMC is able to detect it. Figure 7.3 shows that modified CBMC found this possible violation. Moreover, CBMC creates a counter example trace which is a program trace that ends in a state which violates the property ( $a==c$ ).

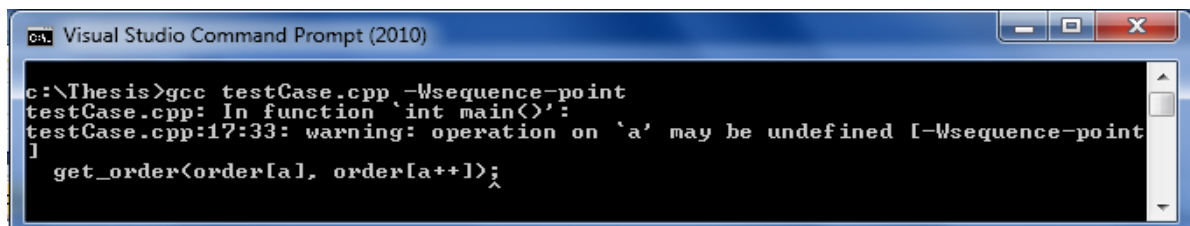


Figure 7.1: GCC 4.8.2

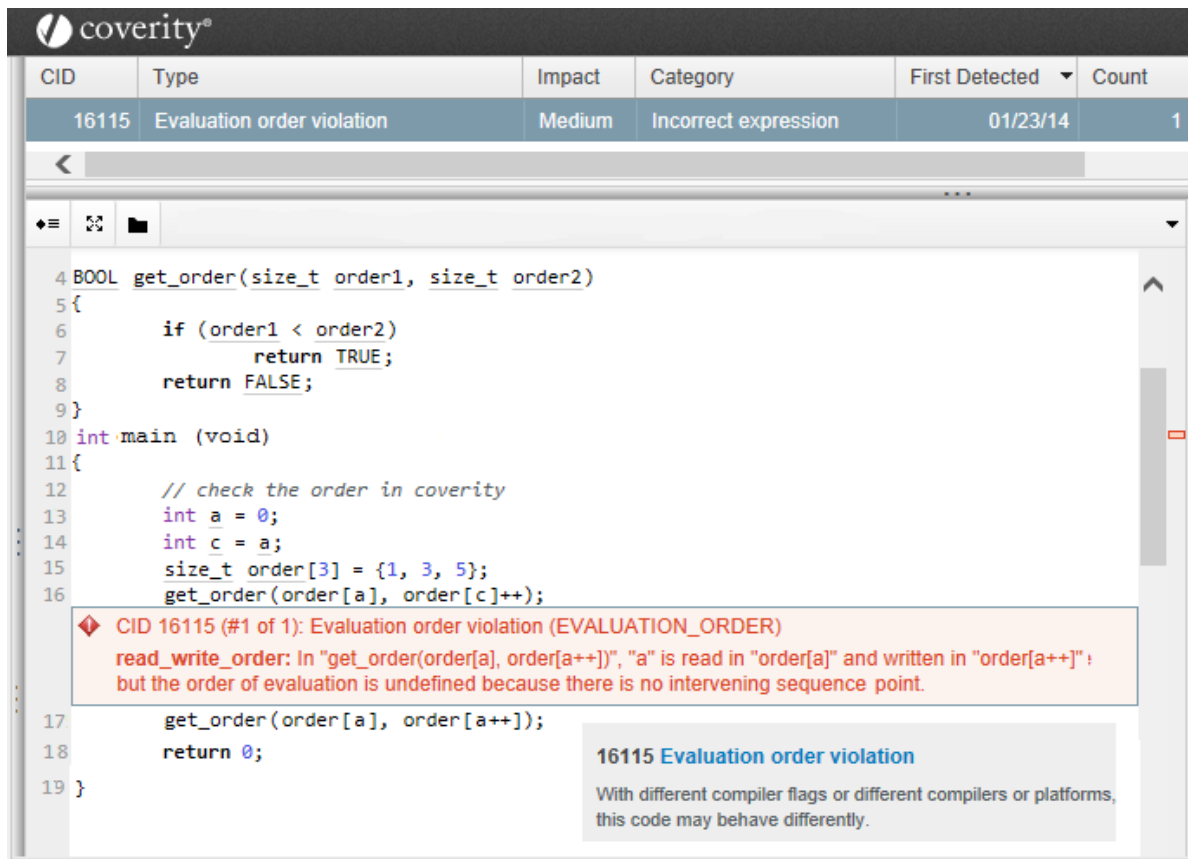


Figure 7.2: Coverity 7.0

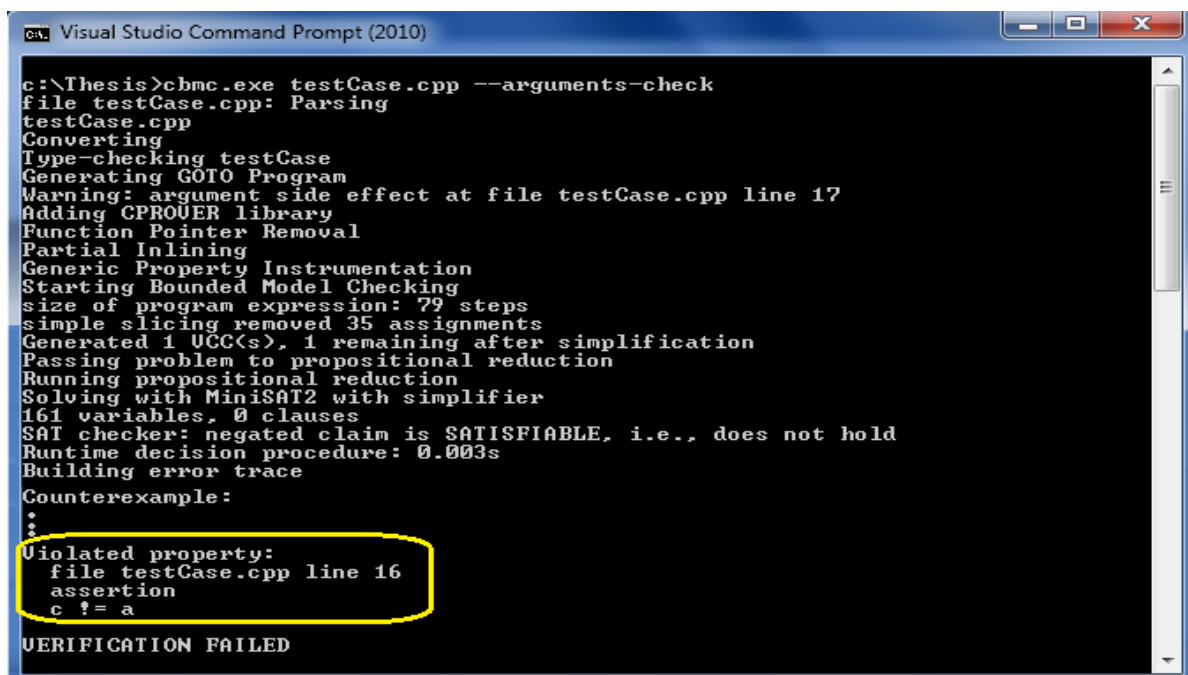


Figure 7.3: Modified CBMC

# Chapter 8

## Related work

There are large amounts of work in both real-time embedded software and checking the quality and correctness of electronic systems. In [14] two main techniques for automatic formal verification of software are presented which are basically focused on tools that provide some form of formal guarantee and aid to improve software quality.

Among them, static analysis techniques are introduced as well-scalable but with a possibly large number of false warnings and limited precision.

The other technique is Model Checking that is presented as an inexpensive technique compared to simulation and testing approaches. It is able to verify specific properties of a system features and automates the verification process fairly well. Specifically, Bounded Model Checkers are very strong at detecting shallow bugs but they are unable to prove properties in the program contains deep loops.

Comparing these two techniques, the model checking-based tools for software are less robust than static analysis tools and the market for these tools is in its infancy. The example tools are Coverity[15] and CBMC, which are famous and powerful static analysis and model checking verification tools, respectively. They find many different defects in any given program.

Detecting unspecified behaviors of a system is certainly an advantage for a verification tool. The order of evaluation of sub expressions is not specified in many programming languages which might cause the side effects and produce different results depending the chosen order of evaluation.

If we are modifying a variable in arguments of a function multiple times without intervening sequence points, it invokes undefined behavior (the comma operator introduces a sequence point but the commas delimiting the function arguments do not). Consider the following case:

```
int i=0;
printf("%d %d %d\n", i++, i++, i);
```

The results might be totally different in:



<b>Operating system</b>	<b>compiler</b>	<b>output</b>
Linux.i686	g++ 4.2.1	1 0 2
	SunStudio	0 1 2
SunOS.x86pc	g++ 4.2.1	1 0 2
	SunStudio	0 1 2
SunOS.sun4u	g++ 4.2.1	0 1 2
	SunStudio	0 1 2

There are many tools that are able to detect some sort of violation in arguments evaluation order. We focused considerably on Coverity, CBMC and GCC to compare them on how good they are able to detect this property in a program.

In Coverity, there are available checkers that traverse paths in a given source code to find specific issues in it. Examples of checkers include EVALUATION\_ORDER, RACE\_CONDITION, RESOURCE\_LEAK, and INFINITE\_LOOP[16].

Coverity is able to check many sorts of side effects in the C/C++ codes and considerably the function argument's evaluation order. It can detect specific instance, where a statement contains multiple side effects on the same value with an undefined evaluation order because, with different compiler flags or different compilers or platforms, the statement may behave differently[16].

GCC can detect violations of this rule when the -Wsequence-point flag is used. It gives an opportunity to receive warning about this issue by turning up the different level of warnings on the compiler (e.g. by -Wall) [10].

CBMC does not really warn this violation to the user and mainly accepts the fact that these behaviors are architecture dependent and it is not desired to show them as they are valid in ANSI-C a not specified [5].

However, these tools, as discussed earlier, partially support the evaluation order defect and they are specially weak at processing expressions including arrays.

# Chapter 9

## Conclusion

During this thesis, we extended the CBMC to verify real-time programs run on FreeRTOS operating system and MDK-ARM firmware and specially found some possible unspecified behaviors.

The targeting program might contain unspecified behaviors, such as, when evaluation order of arguments to a function are not defined by the standard as it depends on many factors like the argument type, the called function's calling convention, the architecture and the compiler. This dependency among expressions could lead to non deterministic behavior of a system and causes serious issues. For this purpose, we prepared a method to detect such an unspecified behavior by extending available tool named CBMC and we equipped a FreeRTOS API to be able to utilize this modification. The CBMC tool was easy to extend and working with it was simple and instructive as it is an open source tool and supported by valuable tutorial and full documentation. Its good reputation and being a notable tool for testing C/C++ programs motivated us to add more supports into it.

In conclusion, we observed our tool worked well at detecting a wide range of different dependencies in expressions of a function's arguments, including direct access to memory locations or through array indexes.

As future work, we could support expressions containing such dependencies, when the pointers of same memory location are involved. It is very similar to cases with arrays that are already included. Further, the current code checks these unspecified behaviors specifically among arguments of any function in a program. We believe the same method is extendable to check them in any expressions in the whole program.



# Bibliography

- [1] E. Clarke and J.M. Wing. *Formal methods: State of the art and future directions*. ACM Computing Surveys (CSUR), 28(4):626-643, 1996.
- [2] D. Kroening. The cbmc homepage. <http://www.cprover.org/cprover-manual/introduction.shtml>, April 2013.
- [3] B. Schlich and S. Kowalewski. *Model checking C source code for embedded systems*. International Journal on Software Tools for Technology Transfer. Volume 11, pp 187-202, July 2009.
- [4] E. Clarke. *Model checking*. In Foundations of Software Technology and Theoretical Computer Science, pages 54-56. Springer, 1997.
- [5] E. Clarke and D. Kroening. *Ansi-C bounded model checker user manual*. Technical report, Technical report, School of Computer Science, Carnegie Mellon University, 2006.
- [6] B. Dirk, H. Thomas A., J. Ranjit and M. Rupak. *The Software Model Checker Blast*. International Journal on Software Tools for Technology Transfer **9** (5-6): 505–525, 2007.
- [7] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata. *Extended static checking for Java*. In Proceedings of the Conference on Programming Language Design and Implementation, pages 234--245, 2002.
- [8] R. Barry. *FreeRTOS Reference Manual - API Functions and Configuration Options*, Real Time Engineers Limited, 2009.
- [9] P. Becker, *Working Draft, Standard for Programming Language C++*, <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf>, 2013.
- [10] *Warning options*, <http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>, 2013.
- [11] Bruno R. Preiss (1998). *Expression Trees*. Retrieved December 20, 2010.
- [12] R. Barry. *Using the FreeRTOS Real Time Kernel - a Practical Guide*, generic CORTEX M3 edition.

- [13] M. Brain, M. Tautschnig. *Beginner's Guide to CPROVER*. March 2014.
- [14] Vijay D'Silva, Daniel Kroening, Georg Weissenbacher. *A Survey of Automated Techniques for Formal Software Verification*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 27, no. 7, pp. 1165–1178, July 2008.
- [15] The Coverity® 6.6 *Deployment Guide*, 2003-2013 Coverity, Inc.
- [16] *Coverity 7.0 Checker Reference*, 2004-2014 Coverity, Inc.