C Bounded Model Checker to Detect Unspecified Expression in FreeRTOS

P.Venkata Gopi kumar
Department of EIE, VNR VJIET, Hyderabad, India
Email: gopikumar_pv@yahoo.com

Abstract—**The 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 paper , it is extended the CBMC tool to check and automatically detect a C/C++ code containing a form of un specified behaviors, like function calla with arguments that exhibits side effects which might be easily un noticed by the programmers. In addition, the code can be configured properly to be used for Arm Cortex micro softwares..**

Index Terms—**Embedded system, CBMC, RTOS, BMC.**

I. INTRODUCTION

The majority of computer devices are embedded systems. These days cell phones, 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. Another 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 this paper, 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 satisfy ability 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

 DOI:10.32377/cvrjst0810

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, it is provide capability for proposed CBMC extension to be run on applications written in FreeRTOS, 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..

This paper 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. Next, author prepared 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 explained why it is essential.

II. EXISTING MODEL CHECKERS

In this section chapter, we briefly introduce important concepts and tools that are used in this work. 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.

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 *CMBC* 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 is shown in Figure 1 [4].

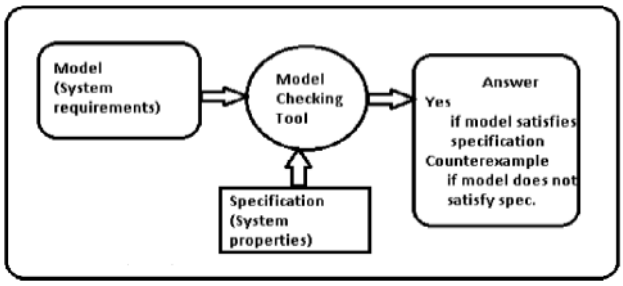

Figure 1: The model-checking approach

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].

III. C BOUNDED MODEL CHECKER (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 SATsolvers. 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.

IV. FREE RTOS 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.

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.

V. UNSPECIFIED SIDE EFFECTS

In this section, it is describe how evaluation of function arguments could be seen as one of the common defects in C/C++. Both expressions give evaluation and defects in formal semantic. In the last part of this chapter, it is explained our algorithm used and show how these side effects in function arguments are detected. 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.

A. 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. 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.

B. 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

 DOI:10.32377/cvrjst0810

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.

In this section, 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:

assumption , requirement
conclusion (*name*)

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

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

C) 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. Your goal is to simulate the usual appearance of papers in a Journal Publication of the CVR College. We are requesting that you follow these guidelines as closely as possible.

VI. IMPLEMENTATION

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

A. 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_vcxprojThe command reads the Makefiles and automatically generates project files for cbmc, gotocc 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 frontends 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 symbolicsimulation 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, gotoprogram or others) are used to create a goto-program, goto-symex to unwind the loops the given number of times and produce and 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 gotocc/ binary. If it is called goto-cc then it emulates GCC flags, goto-arm cc 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:

--no-assertions ignores user assertions
--bounds-check adds array bounds checks
--div-by-zero-check adds division by zero checks
--pointer-check adds pointer checks
--*arguments-check** 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

B. 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

DOI:10.32377/cvrjst0810

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.

VII. EVALUATION AND CASE STUDY

This section summarizes the result of model-checking performed some codes containing undefined behavior in their argument list of functions. Experimented with 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, injected them in separate functions.

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

After testing the code by the mentioned tools, it is 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, 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, the modified CBMC is able to detect it. Figure 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). andFigure shows the GCC 4.8.2.*

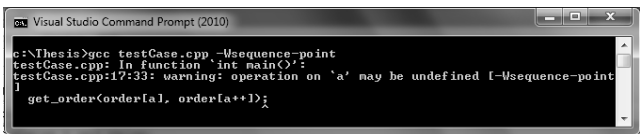

Figure 2: GCC 4.8.2

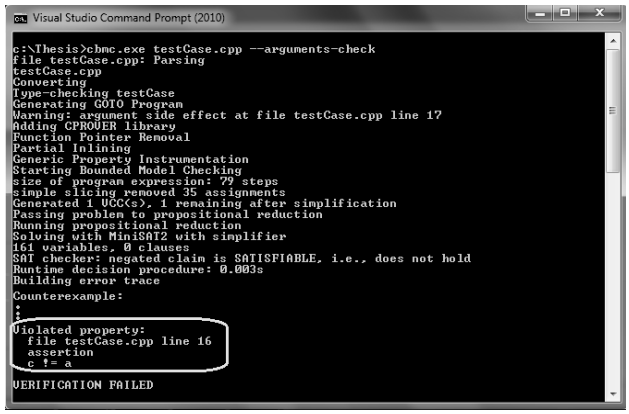

Figure 3: Modified CBMC

VIII. CONCLUSIONS

In this work, author 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, a method is prepared 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, it is observed that, the proposed 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, it 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. The same method can be extendable to check them in any expressions in the whole program.

REFERENCES

[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/cprovermanual/ 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.openstd. org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf , 2013.

[10] *Warningoptions*, 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. 37

[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.38