# A Comparative case study to identify Programming Bugs in ANSI-C based system using Formal Software Verification

Tanjia Akter (Author), University of Asia Pacific, Dhaka, Bangladesh, Email: a.tanjia@yahoo.com.

**Abstract**— Software is the most complex part of today's safety critical embedded systems. Most embedded systems are legacy designs those are written in a low level language such as ANSI-C or even assembly language. Conventional testing methods often fail to detect hidden flaws in complex embedded software such as device drivers or file systems. This deficiency incurs significant development and support/maintenance cost for the manufacturers. Model checking techniques have been proposed to compensate for the weaknesses of conventional testing methods through exhaustive analyses. It has become very demandable way to find those hidden bugs in program and correct the software. Whereas conventional model checkers require manual effort to create an abstract target model, modern software model checkers remove this overhead by directly analyzing a target C program, and can be utilized as unit testing tools. In my research, we present a comparative analysis and the applicability of model checking tools for software in embedded systems.

**Keywords—** Bounded model checking (BMC), Model checking, Software verification, Predicate abstraction.

----------<>----------

## 1. INTRODUCTION

**MODEL** checking[3] is an automatic technique for verifying behavioral properties of a model of a system by exhaustively enumerating its states. It has proven to be a successful technology to verify real-time embedded and safety-critical[1] systems.

Software errors can cause large amounts of damage, not only in safety-critical systems, but also in industrial applications. Software rapidly grows very complex and code reuse introduces code with side-effects unknown to the programmer. In concurrent software, the problems are further intensified because the environment strongly influences the order in which parts of the program are executed, which introduces a source of variation that makes it hard to find a failure. And moreover: it is even hard to reproduce a failure once one is found. Therefore, it is necessary to find automated methods to verify if software fulfills its specification and, if not, to help the user by reporting how the specification was violated. Model checking method is such an automated method to detect the violation.

Currently there are 13 model checking tools with a number of different capabilities suited to different kinds of problems. These tools are BLAST, CBMC, BOOP, SLAM, SATABS, ZING etc. that takes C/C++ as modeling language to verify. In my research work I picked up three popular tools from the existing model checker tool and with some model C program will develop a behavioral comparison between them like the functionalities and techniques used for model checking and other.

To conduct my research I have chosen model checker tool based on Boolean Satisfiability Problem (SAT) tool over Satisfiability modulo theories (SMT). SMT based BMC for software[7] [8] [9] only the theories of uninterrupted functions, arrays and linear arithmetic were considered. The tools that I have used in my research are CBMC, SATABS and ZING. Without these there are other tools that I have tried (such as BOOP, BLAST, MAGIC, etc) but due to some major issues couldn't successfully run those tools. One common issue was target OS doesn't match as my target OS is Windows and there are several tool exists which have no windows supported version. The target property that I have exercised without target tools is user-specified assertions properties because this is the common property which is supported selected tools.

### A. SECTION DETAILS

In section 1, we included the motivation of our thesis work and introduced about model checking and C based model checker tools.

In Section 2, we presented background study of such model checking tools, those are able to verify ANSI C.

In section 3, we discussed about exercised model checking tools such as CBMC which is capable of verifying almost full ANSI. SATABS, which is a bit-precise software model checker for ANSI-C programs and ZING, which is a framework for software model-checking.

In section 4, we added the analyzed result such as findings during experiment of CBMC, SATABS and ZING. We have also added the sample codes runtime data analysis based on different model checker tools that we have used in our research.

Finally in section 5, the conclusion or author opinion is added based on the experience gather from different model checker tools during the research. Tools that are mainly mentioned are CBMC, SATABS and ZING.

## 2. BACKGROUND STUDY

Here, the 13 existing model checkers for ANSI C code are described. Following, we describe two different approaches to model check ANSI C code. There are four approaches to model check ANSI C code[5]. These are –

- Bounded model checking
- Predicate abstraction
- Translate into the model of a general-purpose model checker
- Translate into machine code

These approaches reflect the most prominent features of the respective C code model checker. Some of the model checkers apply more than one of these approaches. All these approaches have to deal with the potentially infinite state space of a C program. A complete overview of C code model checkers known to us is given in the following Table:

Table 1: List of C source code model checkers

| Model checker | Institute | Model | Techniques used |
|---|---|---|---|
| **BLAST** | UC Berkeley | C | CIL, control flow automation, predicate abstraction, CEGAR, theorem prover. |
| **BOOP** | IST Graz | C | Boolean program, predicate abstraction, CEGAR, theorem prover, model checking with MOPED. |
| **CBMC** | CMU | C/C++ | Bounded model checking using SAT solver |
| **FEAVER** | Bell Labs | C | Translation into Promela, model checking with SPIN. |
| **FOCUSCHECK** | Lowa State University | C | CIL, translation into pushdown system, use of constraint solver, model checking of pushdown system |
| **F-SOFT** | NEC | C | CIL, CFG, predicate abstraction, SAT solver, model checking with Verisol(Diver) |
| **MAGIC** | CMU | C | CIL, modular verification, control flow automation, predicate abstraction, CEGAR, SAT solver, model checking with SMV. |
| **MOPS** | UC Davis, UC Berkeley | C | CFG, translation into pushdown automation, model checking of pushdown automation. |
| **SATABS** | CMU | C/C++ | Boolean program, predicate abstraction, CEGAR, SAT solver, model checking with SMV. |
| **SLAM** | Microsoft Research | C | Boolean program, predicate abstraction, CEGAR, theorem prover, model |

| Model checker | Institute | Model | Techniques used |
|---|---|---|---|
| | | | checking with BEBOP. |
| **STEAM** | University Dortmund | C++ | Translation into machine code, state pace generation with Internet C++, Virtual Machine. |
| **ZING** | Microsoft Research | C | Translation into ZING, model checking with ZING model checker. |
| **REDLIB** | Microsoft Research | C/C++ | Library for the model-checking of communicating timed automata's with BDD-like diagrams, TCTL model-checker with timed fairness quantifications, fair simulation checker, and interactive symbolic simulator. |

## 3 EXERCISED MODEL CHECKER TOOLS

From those 13 model checker tools we have worked with three tools those are CBMC, SATABS, ZING. Details of these three model checker tools have given below:

### 3.1 CBMC- C BOUNDED MODEL CHECKER

CBMC is one of such model checking tools, which is capable of verifying almost full ANSI C. It was first developed at CMU by Daniel Kroening and Ed Clarke[11]. It is capable of verifying buffer overflows, pointer safety, exceptions and user-specified assertions. CBMC implements a technique called Bounded Model Checking (BMC), where it convert the program and properties into Boolean formula and SAT solver is used to show whether the formula is satisfiable or not. So if any violated property exists then it will return a counterexample with tracing information, which confirms verification for the safety issues of embedded system.

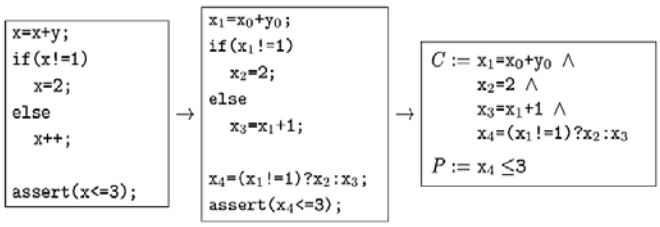



Figure 1: CBMC Program Transformation into a Mathematical Model

### 3.2 SATABS – PREDICATE ABSTRACTION

SATABS[14] is a bit-precise software model checker for ANSI-C programs. It implements sound predicate-abstraction based algorithms for both sequential and concurrent software. It is a verifier for C programs that uses counterexample-guided abstraction refinement based on predicate abstraction as pioneered by SLAM. It uses predicate abstraction technique to find the bug. Predicate abstraction is a method of synthesizing the strongest inductive invariant of a system expressible as a Boolean combination of a given set of atomic predicates. It is a technique commonly used in software model checking in which an infinite-state system is represented abstractly by a finite-state system whose states are the truth valuations of a chosen set of atomic predicates. It has recent successes in software verification. Steps of this technique have given below:

**Sample code** **Predicates**

*int main()* *P1* ⇔
*i=1*
*{* *p2* ⇔
*i=2*
*int i;*
*p3* ⇔ *even(i)*
*i=0;*
*while(even(i))*
***Basic Block**: i++*
*{*
*i++;* ***< I'=i+1 > =***
***T***
*}*
*}*

| $p_1$ | $p_2$ | $p_3$ | | $p'_1$ | $p'_2$ | $p'_3$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | → | 0 | 0 | 0 |
| 0 | 0 | 1 | | 0 | 0 | 1 |
| 0 | 1 | 0 | | 0 | 1 | 0 |
| 0 | 1 | 1 | | 0 | 1 | 1 |
| 1 | 0 | 0 | | 1 | 0 | 0 |
| 1 | 0 | 1 | | 1 | 0 | 1 |
| 1 | 1 | 0 | | 1 | 1 | 0 |
| 1 | 1 | 1 | | 1 | 1 | 1 |

T ^ T ^ T ^ T ^ T ^ T ^ T
=T ^ T ^ T ^ T
=T ^ T
=T
So 2nd iteration is True.

**Property**

*i ≠ 1 ^ i ≠ 2 ^ ¬ even(i) ^ (i'=i+1) ^ i' ≠ 1 ^ i' ≠ 2 ^ even(i')*

| $p_1$ | $p_2$ | $p_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

→ ? →

| $p'_1$ | $p'_2$ | $p'_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

T ^ T ^ T ^ F ^ T ^ T ^ T
=T ^ F ^ T ^ T
=F ^ T
=F
So 1st iteration is False.

**Property**

$i \neq 1 \wedge i \neq 2 \wedge \neg even(i) \wedge (i'=i+1) \wedge i' \neq 1 \wedge i' \neq 2 \wedge \neg even(i')$

Like both iterations, we can abstract the predicates of a program. If all are **True**, then it will terminate again if **False**, then it will generate spurious counterexample.

### 3.3 ZING – PREDICATE ABSTRACTION

Zing is a framework for software model-checking. It supports key programming language constructs such as objects, functions, threads, channels, dynamic allocation. It enables easier extraction of models from code. It supports conformance checking[17]. The Zing language is designed to create executable models of concurrent software, which can be analyzed by the Zing model checker. Properties to check are expressed by assertions. The Zing model-checker is capable of detecting a number of different errors in a Zing model such like as stuck states, Assertion failures and Execution failures. When the zing compiler translates a zing model into a zing object model, then that can be executed to produce transitions between zing states like stacks, global storage and heap. Moreover, the zing state explorer executes the zing object model to explore the state space of the corresponding zing model. The model verification technique is given below:

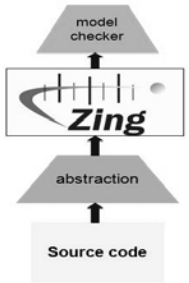



Figure-1: Verifying Model Using Zing

## 4 EXPERIMENTS AND RESULTS

During experiment we got some findings about the model checker tools and comparative results using Binary search and Bubble sort programs. This are-

### 4.1 FINDINGS OF CBMC:

a. It is capable of processing almost full ANSI C.
b. CBMC allow setting the entry point.
c. Provides error tracing information
d. Allow unwind a loop for user desired times
e. Allow to check particular module from a source file
f. Along with other property checking (e.g. Array bounds, pointer safety, exceptions) it also allows assertion checking to which is mostly used in embedded system to trace the error.
g. CMBC can only be used to find errors and not to prove the correctness.

### 4.2 FINDINGS OF SATABS:

a. It is capable of processing almost full ANSI C.
b. SATABS allows to set the model checker name.
c. It sets maximum number of refinement iterations.
d. SATABS allow to set the entry point.
e. Provides error tracing information.
f. It uses heuristic to detect loops.
g. Doesn't allow unwind a loop.
h. Along with other property checking (e.g. Array bounds, pointer safety, exceptions) it also allows assertion checking to which is mostly used in embedded system to trace the error.

### *4.3* FINDINGS OF ZING*:*

a. It is capable of processing object models.
b. ZING allows assertion checking, so all properties to check are expressed by assertions.
c. Check for errors in sets of web services.
d. Check web services for conformance with behavioral contracts.
e. Check behavior of Windows device drivers under concurrent execution.
f. Find errors in complex application protocols.
g. Can query how many processes are in the state.
h. Can "execute" a particular process in the state for one atomic step and return the resulting state.
i. Can compare if two states are equal.

j. Explore the state space of the extracted model.

## 5. COMPARATIVE RESULTS

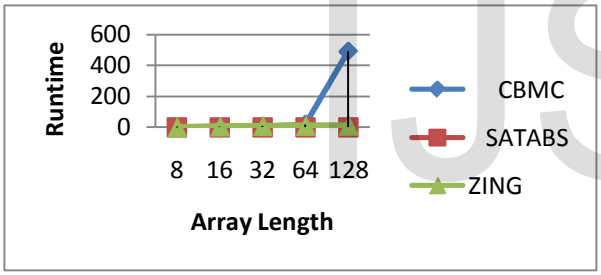
In Binary search I have introduced the following assert condition for an array length 8 and the experiment is done for array length 8, 16, 32, 64, and 128.

**assert((result!=-1&&array[result]== search_item)||(result==-1 && (array[0]!= search_item && array[1]!= search_item &&array[2]!= search_item && array[3]!= search_item && array[4]!= search_item && array[5]!= search_item && array[6]!= search_item && array[7]!= search_item)));**

And the experimented data based on different model checker tool is presented in table 1 has given below.

**Table 1: Comparison table for Binary search**

| | | | | | | |
|---|---|---|---|---|---|---|
| **CBMC** | Array Length | 8 | 16 | 32 | 64 | 128 |
| | Runtime | 0.232s | 0.599s | 2.533s | 18.163s | 490.946s |
| **SATABS** | Array Length | 8 | 16 | 32 | 64 | 128 |
| | Runtime | 0.075s | 0.154s | 0.217s | 0.44s | 1.143s |
| **ZING** | Array Length | 8 | 16 | 32 | 64 | 128 |
| | Runtime | 0.9994s | 7.5753s | 10.3629s | 11.6272s | 12.6575s |




**Graph - 1**

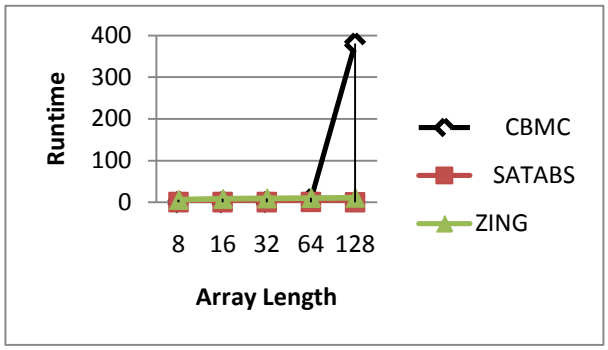
Same way In Bubble sort code we have introduced the following assert condition for an array length 8 and the experiment is also done for array length 8, 16, 32, 64, and 128.

**assert(array[0]<=array[1]&&array[1]<=array[2]&&array[2]<=array[3]&&array[3]<=array[4]&&array[4]<=array[5]&&array[5]<=array[6]);**

And the experimental result based on different model checker tool is presented in table 2.

**Table 2: Comparison table for Bubble Sort**

| | | | | | | |
|---|---|---|---|---|---|---|
| **CBMC** | Array Length | 8 | 16 | 32 | 64 | 128 |
| | Runtime | 0.011s | 0.055s | 0.491s | 6.811s | 380.595s |
| **SATABS** | Array Length | 8 | 16 | 32 | 64 | 128 |
| | Runtime | 0.052s | 0.085s | 0.206s | 0.406s | **UNABLE** |
| **ZING** | Array Length | 8 | 16 | 32 | 64 | 128 |
| | Runtime | 6.0238s | 8.3329s | 9.6148s | 10.6377s | 11.1105s |




**Graph - 2**

For both cases SATABS is taking less time where ZING is taking much more time compare with 2 other tools. Compare with others SATABS is much faster, the reason could be in SATABS it drops the unused function before verification procedure, which make the verification procedure run time efficient. However in bubble sort SATABS was **UNABLE** framework was not able to do the verification for an instance of array of length 32 128, here **UNABLE** means that the framework is unable to validate the program (either because a lack of expression power or time overflow). So in terms of decision procedural time and length of array CBMC indicates the best fitted framework for verification.

## 6. CONCLUSION

In this research different model checkers are discussed and experimented result is shown for sample C code. For this experiment I avoided embedded C code because none of them is currently able to model check C code for embedded systems. Most of them deal with the verification of drivers or protocols written in ANSI C code and are not intended to model check software for embedded systems.

I have described CBMC, SATABS and ZING in this paper. In my research, I tried to use some other tools. But because of some inconvenience like supporting platform, properties etc. I could not go forward with those tools.

I gathered some experimental results with the program that I used in the case study. According to the result and in terms of decision procedural time and length of array CBMC indicates the best fitted framework for verification.

## ACKNOWLEDGMENT

At first the greatest thanks goes to my honorific supervisor, **Nahida Sultana Chowdhury**, who led me into

the interesting areas of Formal software verification methods. I sincerely appreciate her infinite patience and foresighted suggestions that guided me through the difficulties. Additionally, I would like to thank Daniel Kroening, the developer of CBMC Model Checker, who answered my questions and gave me encouragement by google group conversation in the problems of my thesis. And I would also like to thank my thesis reviewer, who approved this thesis report.

## REFERENCES

[1] Edmund Clarke, Daniel Kroening, and Flavio Lerda: **A Tool for Checking ANSI-C Programs.** Carnegie Mellon University.

[2] Rohrbach,M.: **An approach for model checking embedded systems software**. Diploma thesis, RWTH Aachen University (2006).

[3] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers", pp. 146–162, in *SPIN*, 2006.

[4] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking", pp. 794–801, in *ICCAD*, 2006.

[5] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, pp. 69–83, 2009.

[6] Schlich, B., Kowalewski, S.: **Model checking C source code for embedded systems.** In: Margaria, T., Steffen, B., Hinchey, M.G. (eds.) Proceedings of the IEEE/NASA Workshop Leveraging. Applications of Formal Methods, Verification, and Validation (IEEE/NASA ISoLA 2005), NASA/CP-2005-212788, pp. 65–77. NASA, Maryland, USA (2005).

[7] Edmund Clarke, Daniel Kroening: **ANSI-C Bounded Model Checker.** School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213, August 13, 2003.

[8] G´erard Basler, Alastair Donaldson1, Alexander Kaiser2, Daniel Kroening2, Michael Tautschnig2, and Thomas Wahl3: **SATABS: A Bit-Precise Verifier for C Programs.** Imperial College, London, United Kingdom2 University of Oxford, United Kingdom3, Northeastern University, Boston, United States.

[9] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani , Jakob Rehof :**Zing: A Systematic State Explorer for Concurrent Software.** Microsoft Research.