Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples

Herbert Rocha, Raimundo Barreto, Lucas Cordeiro, and Arilo Dias Neto

Federal University of Amazonas
Av. General Rodrigo Octávio Jordão Ramos, 3000
Campus, Coroado I - Manaus/Amazonas
{herberthb12,lucasccordeiro}@gmail.com
{rbarreto,arilo}@dcc.ufam.edu.br
http://portal.ufam.edu.br

Abstract. One of the main challenges in software development is to ensure the correctness and reliability of software systems. In this sense, a system failure or malfunction can result in a catastrophe especially in critical embedded systems. In the context of software verification, bounded model checkers (BMCs) have already been applied to discover subtle errors in real projects. When a model checker finds an error, it produces a counter-example. On one hand, the value of counter-examples to debug software systems is widely recognized in the state-of-the-practice. On the other hand, model checkers often produce counter-examples that are either too large or difficult to be understood mainly because of the software size and the values chosen by the respective solver. This paper proposes a method with the purpose of automating the collection and manipulation of counter-examples in order to generate new instantiated code to reproduce the identified error. The proposed method may be seen as a complementary technique for the verification performed by state-of-the-art BMC tools. In particular, we used the ESBMC model checker to show the effectiveness of the proposed method over publicly available benchmarks and, additionally, a comparison with the tool Frama-C.

1 Introduction

Building complex software systems has been a great challenge to software engineers. This situation can become worse when such software system belongs to a critical embedded system (e.g., aeronautics, space, automotive, health applications) that has to be formally verified to identify errors that may result in failures during the software execution. Thus, verification techniques and software testing are indispensable items for high quality software development.

In the last few years, we can observe a trend towards the application of formal verification techniques to the implementation level. Bounded model checking (BMC) is going to this direction since it has been applied to reason about low-level ANSI-C programs, usually checking safety and/or liveness properties,

considering single- and multi-threaded applications [5, 6]. BMCs have gained popularity due to their ability to handle the full semantics of actual programming languages, and to support the verification of a rich set of properties such as shared variables and locks, arithmetic under- and overflow, pointer safety, array bounds, deadlocks, and fixed-point arithmetic [5].

This paper proposes a method called EZProofC that aims to apply a software bounded model checker, in this case ESBMC (*Efficient SMT-Based Context-Bounded Model Checker*), with the purpose of verifying critical parts of a software written in the C programming language and, additionally, collecting data to show the evidence that failures might happen. ESBMC is a state-of-the-art symbolic context bounded model checker, which performs comparable to other off-the-shelf software model checkers (e.g., CBMC, SATABS) [5]. The motivation of this work is that data collected by verification tools is usually not trivial to be understood, mainly due to the amount of variables and values involved in the counter-example as well as the lack of a standard output to represent the counter-example. The proposed method uses the data provided by counter-examples to generate new instantiated code to reproduce the identified error. In this paper, the instantiated code is a particular instance of the code with the variable values provided by the BMC, which are enough to reproduce the error. This work thus proposes a method where developers can confirm the results provided by the bounded model checker, and additionally, alleviates the process of analyzing large counter-examples, as well as counter-examples that do not characterize an error (i.e., a spurious counter-examples). We adopted the C programming language since it is the standard language to implement several kinds of software including performance-critical software [11]. However, our techniques can also be applied to other programming languages like C++ and Java.

We show the effectiveness of the proposed method over publicly available benchmarks and, additionally, a comparison with the tool Frama-C [4]. Our experimental results show that EZProofC is able to automatically reproduce all failures found in the benchmarks by the adopted BMC tools through the instantiation of the code. Additionally, EZProofC shows a great advantage in comparison to Frama-C since we do not need to write specifications (i.e., pre- and post-conditions) in the source code. We advocate that automating the data collection process we may disseminate the application of formal methods, help developers not very familiar with this subject, and consequently help them to verify more complex C programs.

2 Context-Bounded Model Checking with ESBMC

Model checking has been used successfully to verify actual software (as opposed to abstract system designs) [3, 5, 12], including multi-threaded applications written in low-level programming languages such as ANSI-C [5]. In context-bounded model checking, the state spaces of such applications are bounded by limiting the size of the program's data structures (e.g., arrays) as well as the number of loop iterations and context switches between the different threads that are explored

by the model checker. In symbolic model checking, a set of verification conditions (VCs) is derived from the (bounded) system, which are then solved using a Boolean satisfiability (SAT) or satisfiability modulo theories (SMT) solver.

ESBMC is a symbolic context-bounded model checker based on SMT solvers, which allows the verification of single- and multi-threaded software with shared variables and locks [5]. ESBMC supports full ANSI-C, and can verify programs that make use of bit-level, arrays, pointers, structs, unions, memory allocation and fixed-point arithmetic. It can reason about arithmetic under- and overflows, pointer safety, memory leaks, array bounds violations, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions.

In ESBMC, the program to be analyzed is modeled as a state transition system $M = (S, R, S_0)$, which is extracted from the control-flow graph (CFG). S represents the set of states, $R \subseteq S \times S$ represents the set of transitions (i.e., pairs of states specifying how the system can move from state to state) and $S_0 \subseteq S$ represents the set of initial states. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to *pc*. ESBMC identifies each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given the transition system M, a safety property ϕ, a context bound C and a bound k, ESBMC builds a *reachability tree* (RT) that represents the program unfolding for C, k and ϕ. ESBMC then derives a *verification condition* (VC) ψ_k^π for each given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^π is satisfiable if and only if ϕ has a counter-example of depth k that is exhibited by π. ψ_k^π is given by the following logical formula:

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{1}$$

where function I characterizes the set of initial states of M and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents executions of M of length i and ψ_k^π can be satisfied if and only if for some $i \leq k$ there exists a reachable state along π at time step i in which ϕ is violated. ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. If ψ_k^π is satisfiable, then ϕ is violated along π and the SMT solver provides a satisfying assignment, from which ESBMC can extract the values of the program variables to construct a counter-example.

A counter-example is a trace that shows that a given property does not hold in the model [1]. Counter-examples allow the user: (i) to analyze the failure; (ii) to understand the root of the error; and (iii) to correct either the specification or the model, in this case, from the property and the program that has been analyzed respectively. A counter-example for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less along π. Finally, we can define $\psi_k = \bigwedge_\pi \psi_k^\pi$ and use it to check all paths.

3 EZProofC Method

This section describes the main steps of the EZProofC method[1] which aims to explore the counter-examples generated by the ESBMC model checker, in such a way that it can generate a new instantiated code to reproduce the errors. It is important to emphasize here that we could adopt any BMC tool.

Figure 1 shows an overview of our proposed method. The EZProofC method consists of the following steps: (i) code preprocessing; (ii) model checking with ESBMC; (iii) generation of a new instantiated code; and (iv) code execution and confirmation of defects.

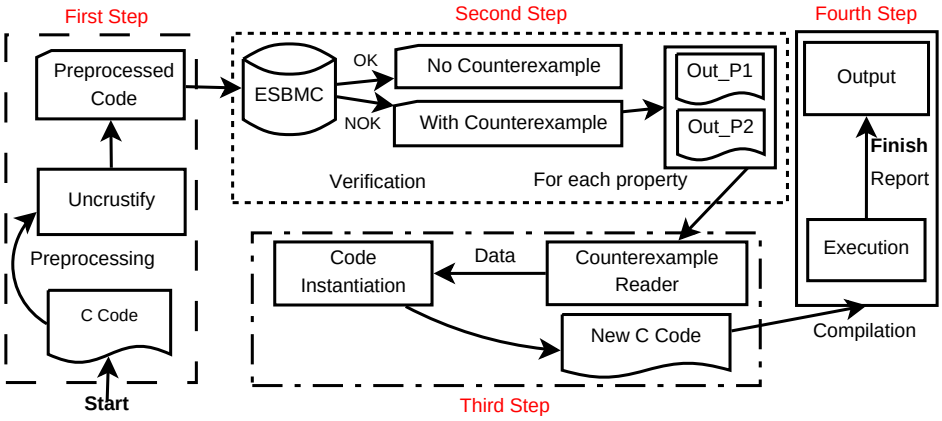

Fig. 1. Flow structure of the proposed method.

To explain the main steps of our proposed method, we use the application Sendmail[2], in particular, the code `tTflag_arr_two_loops_bad.c` extracted from the Verisec[3] benchmark suite, which is the the standard Unix mail (SMTP) server. This code has 64 lines of code and aims to parse a string of digits into two signed integers.

3.1 First Step: Code Preprocessing

In the first step, the analyzed code is preprocessed using the tool *UNCRUSTIFY*[4] that will preprocess the code, as show in Figure 2, to define a standard formatting to improve the presentation of the formatting items such as: indentation, block delimiters, one command per line, delineation of structures, and other formatting aspects. This preprocessing step allows a better identification of structures contained in the code, facilitating its handling and making it easier to implement the next steps. It is important to note that Figure 2 presents just a fragment from the original code.

[1] Available at https://sites.google.com/site/ezproofc/
[2] Available at http://www.sendmail.org
[3] Available at http://se.cs.toronto.edu/index.php/Verisec_Suite
[4] Available at http://uncrustify.sourceforge.net

```
1  #define INSIZE 14
2  int main (void){
3  unsigned char in[INSIZE+1];
4  unsigned char c;
5  int i, j;
6  int idx_in = 0;
7  ...
8  /*accumulate last(int) from in (char[])*/
9  c = in[idx_in];
10 if (c == '-')
11 {
12  i=0;
13  idx_in++;
14  c = in[idx_in];
15  while (('0' <= c) && (c <= '9'))
16  {
17    j = c - '0';
18    i = i * 10 + j;
19    idx_in++;
20    c = in[idx_in];
21   }
22 }
23 }
```

Fig. 2. C code fragment already pre-processed.

3.2 Second Step: Model Checking with ESBMC

In the second step, we use the ESBMC to verify the properties that are violated in the code. ESBMC divides the verification in two levels: In the first level, ESBMC determines which properties might be violated by means of preliminary static analysis (using abstract interpretation), for determining program locations that potentially contain an error (these properties are called claims). It is worth to note that claims are automatically generated by ESBMC. Due to the imprecision of the static analysis, there is the need to go the second level, that is, ESBMC has to confirm that these claims are indeed genuine errors by using a more complete and accurate verification technique (it is important to emphasize that during the verification, ESBMC adopts the program slicing technique [14]).

The verification result may be classified in two ways: the code was checked and there is no counter-example (i.e., the property was verified but no error has been found up to the given bound k) and the code was verified and there is a counter-example (i.e., a property violation has been found, as shown in Figure 3) which presents the violation of the property "`idx_in<15`" identified in the code fragment shown in Figure 2 (line 20). Additionally, for the verification process, ESBMC has an Eclipse plug-in, [5] which allows the user to locate the variable in the counter-example directly in the analyzed code. To explain clearly each proposed step, we decided to analyze one specific `claim` as shown in line 20 of Figure 2.

The property "`idx_in<15`" has been violated due to the fact that in the array index `in` the variable `idx_in` exceeds the upper bound of the array `in` as

[5] Available at http://www.eclipse.org

```
Counterexample:

(.....)
State 55 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 9 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::c=45 (00101101)

State 58 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 13 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::idx_in=9 (00000000000000000000000000001001)

State 59 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 14 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::c=48 (00110000)

State 96 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 17 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::j=3 (00000000000000000000000000000011)

State 97 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 18 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::i=33 (00000000000000000000000000100001)

State 98 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 19 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::idx_in=15 (00000000000000000000000000001111)

State 93 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 20 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::c=51 (00110011)

Violated property:
  file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 20 function main
  array `in' upper bound
  idx_in < 15

VERIFICATION FAILED
```

Fig. 3. Counter-example.

defined in line 3 (`in[INSIZE+1]`) of Figure 2, where `INSIZE` is defined with the value 14. As the loop in line 15 does not control the value of the variable `idx_in`, in state 98 this variable receives a value greater than the upper bound of the array `in`, which thus causes the UPPER BOUND violation.

3.3 Third Step: Code Instantiation

The third step is divided into two phases: analysis of counter-examples produced in step 2 and generation of a new instantiated C code. Algorithm 1 details the method to run both phases. The runtime complexity of this algorithm is $O(n + m)$, where n is the size of the analyzed C code and m is the size of the counter-example. The inputs of this algorithm are the analyzed code (`Code`) and the counter-example (`CE_Out`). Initially, the counter-example (`CE_Out`) is analyzed to collect several pieces of information, such as: (1) the variables involved in the property violation; (2) the line number where values are assigned to variables; and (3) the specific value for each variable. This information is obtained by the function `GetValuesCEER` (line 1 of the Algorithm 1) through regular expressions applied to the counter-example file. This function returns a set that contains data about the variables found in the counter-example (e.g., `Var{vline = 9,var = c,vvalue = 45}`), the violated property (`P`) and the line number where the property has been violated (`line_p`).

In this way, the analyzed code is read (starting from line 8), as well as the counter-example. If the line number of the variable identified in the counter-

```
   Input: Code, CE_Out
   Output: New instantiated code
   // first phase
 1 Var,P,line_p ← GetValuesCEER(CE_Out);
 2 SCE ← {Var{vline,var,vvalue},P,line_p};
 3 size ← GetTotalLineCE(SCE[Var[]]);
 4 Lines, tline ← GetValuesCode(Code);
 5 SCode ← {Lines{ }, tline};
 6 UPCASE ← {Set of specific cases for counter-example data collection};
 7 i, k ← 1;
   // second phase
 8 while i ≤ SCode[tline] do
 9     if i == SCE[Var[vline[k]]] AND k ≤ size then
10         if SCE[P] OR SCE[Var[vvalue[k]]] ∈ UPCASE then
11             New_Line ← StartTrigger(SCE[P], SCE[Var[vvalue[k]]]);
12             WriteLineCode(New_Line); k ← k + 1;
13         end
14         else
15             New_Line ← "SCE[Var[var[k]]] = SCE[Var[vvalue[k]]]";
16             WriteLineCode(New_Line); k ← k + 1;
17         end
18     end
19     else
20         WriteLineCode(SCode[Lines[i]]); i ← i + 1;
21     end
22 end
```

Algorithm 1: Counterexample2NewCode

example is equal to the line number of the analyzed code we can generate a new line of code; where the identified variable receives the value abstracted from the counter-example (e.g., the following values of the variables gathered from the counter-example `line = 9, var =c` and `value = 45` result that the variable New Line (in line 15) receives the text c = 45, which thus generates a new line). Importantly, the instantiation of the variables in the new code is strictly executed according to the sequence in the counter-example. For instance, if the same variable in the counter-example is mentioned multiple times in the same line (for example, in loops), only the last value found in the counter-example will be assigned to the variable in the instantiated code.

Improving the counter-example data collection, the proposed method may require a separate approach for some specific cases, where it is applied to the verification step of the EZProofC method (see Section 3.2) or triggered by the analysis of the counter-example. Line 10 of the Algorithm 1 checks whether the property or a variable in the counter-example is in a set of specific cases already predefined (line 6 variable `UPCASE`). Thus, if there is some specific case in the counter-example that has been identified, the proper approach is applied by the adoption of the function `StartTrigger` in the line 11, as following:

(i) When the violation of a property is identified and there is not enough information about the counter-example, it is necessary to use in the verifica-

tion step with ESBMC, particularly in smaller code, the option `--no-slice` which does not remove unused equations of the program for generating the counter-example. Another way to diversify the values of variables, and hence the result in the counter-example, is to apply non-deterministic values to them, e.g. `a[0]=nondet_int()`;

(ii) In some specific cases, the violation of the property UPPER BOUND can generate a counter-example without the data about the upper bound of the array. In this case, the method firstly identifies the array name and, through the analysis of the code, it can collect the upper bound. This procedure is accomplished by two elements: the first is the function `NUM_OF(arr)` to get the array size; and the second element is an `assert` that will contain the result of the function `NUM_OF(arr)` and the index value of the array that was identified in the counter-example, thereby the structure of the assert will be the following `assert((N)<=NUM_OF(arr)-1)`, where `N` is the index value, that will be adopted to validate the bound of the array;

(iii) Considering dynamic memory allocation violations, the proposed method has to analyze: (1) if the pointer is referencing to the correct object; (2) if the pointer points to an invalid object; (3) if the object considered is a dynamic object; and (4) the argument of a `free` function call if in the deallocation procedure is still a valid pointer value. The aim of this analysis is to obtain a right assertion about the property identified.

The second phase of this third step from the EZProofC method aims to generate a new instantiated code. The method only makes a copy of the original code (in line 20 of the Algorithm 1), and replaces variables assignments using the specific values identified in phase one (in line 12 or 16). In the case of properties such as `UPPER BOUND` or `LOWER BOUND`, the proposed method includes assertions in the instantiated code to reproduce the error, as mentioned before in line 11 about the triggers in the analysis of the counter-example. Such assertions contain data from the property identified in the counter-example. The final result of this step is an instantiated C code with the values of variables that are extracted from the counter-example, as shown in Figure 4. It is worth noting that in the counter-example (see Figure 3), the property violated was `UPPER BOUND`, and the data was "`idx_in<15`". In this case, in line 20 of Figure 4, the proposed method has included an assertion in order to reproduce the error.

In particular, in this example it is obvious that the assertion will fail. This is because the previous instruction assigns exactly a value that contradicts the assertion. However, it is worth observing that this assignment comes directly from the counter-example, implying that there is a situation where this assignment happens in one of the execution paths.

3.4 Fourth Step: Code execution and confirmation of errors

In the third step of this method, we generate one instantiated program for each property violated. In this fourth step, each instantiated code is compiled and executed. The result of the execution demonstrates the error (`Line:20:main: Assertion & 'idx_in<15' failed. Aborted`) pointed out by the counterexample.

```
1  #define INSIZE 14
2  int main (void){
3  unsigned char in[INSIZE+1];
4  unsigned char c;
5  int i, j;
6  int idx_in = 0;
7  ...
8  /*accumulate last(int) from in (char[])*/
9  c =45 ; //<- by EZProofC
10 if (c == '-')
11 {
12  i =0 ;
13  idx_in = 9 ; //<- by EZProofC
14  c =48 ; //<- by EZProofC
15  while (('0' <= c) && (c <= '9'))
16  {
17    j =3 ; //<- by EZProofC
18    i =33 ; //<- by EZProofC
19    idx_in = 15 ; //<- by EZProofC
20    assert(idx_in<15); //<- by EZProofC
21    c =51 ; //<- by EZProofC
22  }
23 }
24 }
```

Fig. 4. C code already instantiated.

4 Experimental Results

This section describes the planning, design, execution, and the analysis of the results of an empirical study conducted with the purpose of evaluating the proposed method when applied to the verification of standard ANSI-C benchmarks and, additionally, a comparison with the tool Frama-C[6] [4] version Boron-20100401. Frama-C is a suite of tools dedicated to the analysis of software written in C. Frama-C makes it possible to observe sets of possible values for the variables of the program at each point of the execution. Frama-C also allows verifying that the source code satisfies a provided formal specification. The specifications can be written in a dedicated language, in this case, ANSI/ISO C Specification Language (ACSL).

The experiments were conducted on an Intel Core 2 Duo CPU, 2Ghz, 3GB RAM with Linux OS. The proposed method was implemented in a tool called EZProofC using the ESBMC v1.16 model checker.

4.1 Planning and Design the Experiments

The goal of this empirical evaluation is to analyze the impact of the proposed method with the purpose of confirming the properties reported by the model checker as possible errors in the code. This confirmation is based on the number of properties (*claims*) reported by the model checker, which should be confirmed by the proposed method.

[6] http://frama-c.com/

In order to evaluate the proposed method, we considered 211 ANSI-C programs from six different benchmarks selected with the aim to evaluate the capacity and performance of methods and techniques in the identification of errors. Moreover, such ANSI-C programs from these standard benchmark suites represent real implementations. The adopted benchmarks were: (i) EUREKA[7] which contains programs that allow us to assess the scalability of the model checking tools on problems of increasing complexity. It is worth observing that some of the programs represent more than one execution, with different input data. For instance, the program `bubble_sort1_13.c` represents 13 instances (from 1 to 13) of the program `bubble_sort.c`. The program `prim4_8.c` represents 5 instances (from 4 to 8) of the program `prim.c`; (ii) SNU[8] which contains C programs used for worst-case execution time analysis, where such programs are mostly of numeric analysis and DSP (Digital Signal Processing) algorithms; (iii) WCET[9] which, in the same way as SNU, contains programs used for worst-case execution time analysis; (iv) NEC[10] which contains C programs that allow us to check error-detection easily since they provide ANSI-C programs with and without known errors; (v) Siemens (SIR[11]) which is a test suite for lexical analyzer, pattern matching and (vi) some ANSI-C programs taken from the CBMC (C bounded model checker) tutorial [12].

During this empirical evaluation, each program of the benchmark was executed using 3 methods: (1) Application of the EZProofC method (see Section 3), i.e., code preprocessing, identification of claims, verification, analysis of counterexamples, and code instantiation; (2) Application of the tool Frama-C with the option `-val`, which means that the *value analysis* plug-in is called in such a way that it computes automatically variation domains for the variables of the program. This plug-in is used to infer absence/presence of runtime errors; and (3) Application of the tool Frama-C with the plug-in `Jessie`, which is a plug-in that allows deductive verification of C programs annotated with ACSL [2]. The *verification conditions (VC)* are verified by the `Z3` theorem prover [13], which is the same standard theorem prover used by the ESBMC model checker. In this way the tool Frama-C was executed as: `frama-c -jessie -jessie-atp=z3 <file.c>`, where `<file.c>` is the C code that will be verified.

4.2 Experiment's Execution and Results Analysis

After executing the benchmarks, we obtained the results shown in Table 1, where each column of this table means: (1) the application identification (ID), (2) the C program name and, additionally, in some cases, the range of instances, e.g., file1_13.c, meaning that there are 13 instances, from 1 to 13. In Table 1 the

[7] http://www.ai-lab.it/eureka/bmc.html
[8] http://www.cprover.org/goto-cc/examples/snu.html
[9] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[10] http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php
[11] http://sir.unl.edu/portal/index.html
[12] http://www.cprover.org/cbmc/doc/manual.pdf
[13] http://research.microsoft.com/en-us/um/redmond/projects/z3/

programs from 1 to 16 come from the EUREKA benchmark, from 17 to 19 come from the CBMC tutorial, program 20 comes from the NEC benchmark, from 21 to 22 come from the SNU benchmark, program 23 comes from WCET benchmark, and program 24 comes from SIR benchmark; (3) the lines of code - LOC (#L); (4) the amount of identified *warnings* (#W) and the execution time (TW) of the Frama-C with the *value analysis plug-in*; (5) the total number of properties (or `claims`) that may be violated (#P), the execution time of the properties identification spent by ESBMC and EZProofC (TC), the execution time of the verification of all properties spent by ESBMC (TV), total number of properties that have been violated and reproduced using the EZProofC method (#V), and the number of lines in the counter-examples (CE); and (6) the number of properties found in common (Same Claims & Warnings) between the EZProofC (claims) and the Frama-C (warnings).

It is important to note that for programs with several instances, the number of violated properties presented is that of the highest instance value. Additionally, in case of programs with more than one instance, the number of lines in the counter-examples (#CE) and properties found in common (Same Claims & Warnings) is respectively the largest counter-example found and the largest number of properties found in common. The results of the application of the proposed method, as well as the EZProofC tool are available at https://sites.google.com/site/ezproofc/.

Table 1. Details related to the execution time of the benchmarks

#	Module	#L	Frama-C		EZProofC/ESBMC					Same Claims
			#W	TW	#P	TC	TV	#V	CE	& Warnings
1	bf5_20.c	49	6	<1s	33	<1s	<60s	0	-	0
2	bubble_sort1_13.c	51	2	<1s	25	<1s	<15s	0	-	0
3	fibonacci1_13.c	25	1	<1s	1	<1s	<1s	0	-	0
4	init_sel_sort1_13.c	54	2	<1s	25	<1s	<15s	0	-	0
5	minmax1_13.c	19	6	<1s	9	<1s	<3s	0	-	0
6	minmax_unsafe1_13.c	19	6	<1s	9	<1s	<4s	1	16	0
7	n_k_gray_codes1_13.c	45	36	<1s	22	<1s	<120s	0	-	11
8	no_init_bubble_sort_safe1_13.c	25	2	<1s	14	<1s	<7s	1	32	1
9	no_init_sel_sort1_13.c	41	5	<1s	25	<1s	<15s	12	144	3
10	no_init_sel_sort_safe1_13.c	28	5	<1s	14	<1s	<7s	1	32	3
11	no_init_sel_sort_unsafe1_13.c	28	5	<1s	14	<1s	<7s	1	32	3
12	prim4_8.c	79	12	<1s	30	<1s	<60s	0	-	3
13	selection_sort1_13.c	54	2	<1s	25	<1s	<15s	0	-	0
14	strcmp1_13.c	15	4	<1s	6	<1s	≈14400s	3	80	0
15	sum1_13.c	21	1	<1s	1	<1s	<1s	1	48	0
16	sum_array1_13.c	11	1	<1s	7	<1s	<3s	1	8	0
17	assert_unsafy.c	15	4	<1s	1	<1s	<1s	1	24	0
18	bound_array.c	16	2	<1s	10	<1s	<10s	1	30	1
19	division_by_zero.c	32	3	<1s	1	<1s	<1s	1	24	1
20	ex26.c	29	4	<1s	8	<1s	≈420s	2	1236	1
21	crc_det.c	125	1	<1s	15	<1s	≈840s	0	-	1
22	select_det.c	122	3	<1s	39	<1s	≈14400s	3	40	1
23	cnt_nondet.c	139	0	<1s	16	<1s	<1s	0	-	0
24	Siemens_print_tokens2.c	508	90	<1s	51	<1s	≈18000s	1	3344	34

As shown in Table 1, the EZProofC method is *scalable* to any code and counter-example size, since the complexity of the proposed method algorithm is $O(n+m)$. The execution time of EZProofC is thus linear, even when considering different code sizes, as we can see in the experiments' execution time.

One could argue that the selected benchmarks may not represent well all the possible scenarios for applying the proposed method, mainly when taking into account the programs size in terms of LOCs. However, as an example consider the experiment with the program 20 from Table 1, which has only 29 LOCs but it was the one that produced some of the largest counter-examples, in this case 1236 lines. Note further that this counter-example has a trace that shows all the variables, as well as the assignments included in a specific execution (i.e., including loops) that will result in the violation of the property that has been identified by ESBMC (i.e., unwind of a specific execution of the program). The drawback of the EZProofC tool is that it relies on the scalability of the adopted model checker, since it depends only on it to generate the counter-examples. Apart from that, the proposed method is able to scale to large sizes of counter-examples, in this case, from 8 up to 3344 lines. However, we believe that the limiting factor on the size of the counter-example is far beyond this.

Analyzing the Frama-C tool with the *value analysis plug-in*, it is important to emphasize that the results about *warnings* (in the column #W) are very effective, providing the user with a good support to explore the code that has been analyzed. However, such *warnings* were not only about safety properties, but involved analysis of the structures of the code (e.g., return of functions). This partly explains why the number of properties between the EZProofC (claims) and the Frama-C (warnings) (column Same Claims & Warnings of Table 1) are rather different.

The Frama-C tool also allows the use of other plug-ins, for instance, the plug-in `Jessie`, which aims to perform deductive verification of C programs not using, in this case, static analysis. The C program does not need to be complete nor annotated to be analyzed with the Jessie plug-in [10]. However, in the experiment conducted, Jessie plug-in did not find any property violation, i.e., no error was found, even though Frama-C pointed out several *warnings*. Jessie plug-in also allows to prove that C functions satisfy their specification as expressed in ANSI/ISO C Specification Language (ACSL). We understand that the verification of Frama-C could be improved by writing such specifications on the analyzed C code. However, the inclusion of such specifications may be hard and error-prone, especially for legacy code. Therefore, if we compare the use of Frama-C/Jessie and the EZProofC, we argue that a great advantage of EZProofC is not requiring such auxiliary specifications. EZProofC is a completely automatic method that does not need to write specifications, and neither preconditions and postconditions. Additionally, in the case of the Frama-C, the user has to act explicitly to reproduce the error using the computed values.

In these experiments some situations need to be pointed out about the application of the EZProofC method.

- In program 20 from Table 1, EZProofC identified properties of safety pointers and dynamic memory allocation (`POINTER_OFFSET` and `SAME-OBJECT`). The property identified was `UPPER BOUND` and the data was `!(2 * y + POINTER_OFFSET(x) >= 200) || !(SAME-OBJECT (x, &b[0]))`. However, after handling all information (see Section 3.3), this resulted only in the following assertive `(2*y + x >= 200) || (x != b)`.
- Program 24 is considered the golden version code (i.e., the supposed correct version). Taking into account that this code is very large, and requires a significant amount of memory, the verification was performed in a function-by-function basis. Particularly, we checked the function `get_token`. The error identified in this code is the `UPPER BOUND` violation of `array buffer`, which is declared with the upper bound of 80. However, based on the proof of the error, it is noticed that the index of this array, the variable `i`, exceeded the upper bound, causing the violation of property $i < 81$, in the same way as identified in the work of Cordeiro *et.al.* [6].

We have shown that the manipulation of the counter-example is not always a trivial task. During the experiments, we obtained relatively large counter-examples (e.g., 3344 lines). However, the application of our proposed method decreases substantially the complexity of this task, i.e., the EZProofC solves the problem in less than 1s (without the verification step with the model checker), to manipulate a large amount of data, variables and their values. It is important to emphasize the need for verifying each property (*claims*) identified in the analyzed code. This is because these properties do not necessarily correspond to errors, but these are only potential failures. This is the reason by which the number of properties identified in Table 1 is greater than or equal to the number of errors reproduced.

5 Related Work

In the technical literature, there are several tools and methods for analysis of counter-examples and debugging code for error-proof. Many studies have addressed this problem (e.g. [4], [7], [9], [13]), that aim to find the root cause of a failure in the model, and propose automated means of extracting more information about the model, facilitating the debugging process.

Ji *et al.* [9] present a software debugger used for finding errors in C programs. In the same way, EZProofC aims to demonstrate errors found by BMCs. The difference is that our technique tests the system exhaustively for verifying that a given property is part of the model. Additionally, BMCs run the code symbolically, that is, they do not test programs with fixed entry values, but create a mathematical model of the program [1]. Debuggers, however only evaluate execution paths that were defined according to the input variables. Thus, a debugger will not exhaustively test the state space of the analyzed code.

Taghdiri and Jackson [13] propose a counter-example guided refinement of an abstraction to check programs written in any programming language that supports procedure declarations and can be translated to logical constraints. In

the same way as our work, they propose a "validity check", where the validity of each behavior in the counter-example is checked in the original program. They use a SAT solver, and our work uses ESBMC that adopts an SMT solver. Nevertheless, if the counter-example is invalid they propose to adopt a "specification inference", that is, the specification is not provided by the user but automatically inferred from the code. In our opinion, the drawback of such method is the limited applicability since they target to solve only structural properties, i.e., properties that constrain the configuration of the heap after the execution of a procedure.

Astrée[14] [7] is a completely automatic analyzer that aims to prove the absence of run time errors (RTE) in C programs. The design of Astrée is based on abstract interpretation, which is a formal theory of discrete approximation. Astrée analyzes structured C programs, with complex memory usages, but without dynamic memory allocation and recursion. It also excludes union types, unbounded recursive functions calls, and the use of C libraries. In the same way as Astrée, the EZProofC aims to produce a correctness proof for complex software without any false alarm (or spurious counter-examples). However, EZProofC differs from Astrée in the sense that the proposed verification is made by a bounded model checker which provides support for structures not supported by Astrée.

6 Conclusions and Future Work

The main purpose of this paper is to help developers not familiar with formal verification techniques to use a model checker tool to find failures in the software and to verify that such errors may happen. We described a method called EZProofC that aims to contribute as a complementary technique to the verification performed by BMCs. Specifically, we have used the ESBMC tool, which is a state-of-the-art symbolic context bounded model checker. Basically, our method proposes to automate the gathering and manipulation of the counter-example generated by ESBMC in order to reproduce the identified error.

The experimental results have shown to be very effective over publicly available benchmarks. In this case, we could reproduce all failures encountered by the adopted BMC tool. On the one hand, we demonstrate that EZProofC has some advantages, when compared to Frama-C, mainly because EZProofC can automatically reproduce the identified property violation, through the generation of a instantiated code. On the other hand, the Frama-C requires the user to act explicitly to demonstrate the error using computed values.

We noticed that due to the state space explosion problem, the user may ask to the BMC to adopt simplifications in the model (e.g. function-by-function verification). In some situations, this can lead to spurious results, i.e., a counter-example may not truly characterize an error. In this way, we want to investigate the inclusion of additional data during the phase of new instantiated code generation in order to demonstrate the verification (with such simplifications). For instance, in the case of verifying a program function-by-function, we need to

[14] http://www.astree.ens.fr/

include the values of variables that are dependent on other functions other than the function being verified. Additionally, we intend to extend our experiments to evaluate the usability of the proposed method. We also plan to adapt the proposed method to use other model checkers (Blast [3] and Java PathFinder [8]) that rely on other abstraction techniques. We think that the adjustment will be in most part on regular expressions, which was the way we implemented data gathering and new code generation.

Acknowledgement

The authors acknowledge the support granted by FAPESP process 08/57870-9, CAPES process BEX-3586/10-3, and by CNPq processes 575696/2008-7, and 573963/2008-8.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. In: CEA LIST and INRIA (2009), `http://frama-c.cea.fr/acsl.html`
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. In: Int. J. Softw. Tools Technol. Transf. (STTT). vol. 9, pp. 505–525 (2007)
4. Canet, G., Cuoq, P., Monate, B.: A Value Analysis for C Programs. In: Intl Conf. on Source Code Analysis and Manipulation (SCAM). pp. 123–124 (2009)
5. Cordeiro, L., Fischer, B.: Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In: Intl. Conf. on Software Engineering (ICSE). pp. 331–340 (2011)
6. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In: IEEE Transactions on Software Engineering (TSE). vol. 99 (2011), `http://eprints.ecs.soton.ac.uk/22291/`
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Programming Languages and Systems (PLS). vol. LNCS 3444, pp. 21–30 (2005)
8. Havelund, K.: Java PathFinder, A Translator from Java to Promela. In: Intl. SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking. p. 152 (1999)
9. Ji, J.H., Woo, G., Park, H.B., Park, J.S.: Design and Implementation of Retargetable Software Debugger Based on GDB. In: Intl. Conf. on Convergence and Hybrid Information Technology (CHIT). vol. 1, pp. 737–740 (2008)
10. Marché, C., Moy, Y.: Jessie plugin tutorial. In: INRIA (2010), `http://frama-c.com/download/jessie-tutorial-Carbon-20101201-beta1.pdf`
11. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: CETS: Compiler Enforced Temporal Safety for C. In: SIGPLAN Notes. vol. 45, pp. 31–40 (2010)
12. Schlich, B., Kowalewski, S.: Model checking C source code for embedded systems. In: Int. J. Softw. Tools Technol. Transf. (STTT). vol. 11, pp. 187–202 (2009)
13. Taghdiri, M.: Inferring Specifications to Detect Errors in Code. In: Intl. Conf. on Automated Software Engineering (ASE). pp. 144–153 (2004)
14. Tip, F.: A survey of program slicing techniques. Journal Programming Languages 3(3) (1995)