Encoding floating-point numbers using the SMT theory in ESBMC: An empirical evaluation over the SV-COMP benchmarks

Mikhail Y. R. Gadelha[1], Lucas C. Cordeiro[2], and Denis A. Nicole[1]

[1] Electronics and Computer Science, University of Southampton, UK
myrg1g14@soton.ac.uk,dan@ecs.soton.ac.uk
[2] Department of Computer Science, University of Oxford, UK
lucas.cordeiro@cs.ox.ac.uk

Abstract This paper describes the support for encoding C/C++ programs using the SMT theory of floating-point numbers in ESBMC: an SMT-based context-bounded model checker that provides bit-precise verification of C and C++ programs. In particular, we exploit the availability of two different SMT solvers (MathSAT and Z3) to discharge and check the verification conditions produced by our encoding using the benchmarks from the International Competition on Software Verification (SV-COMP). The experimental results show that our encoding based on MathSAT is able to outperform not only Z3, but also other existing approaches that participated in the most recent edition of SV-COMP.

Keywords: Floating-Point Arithmetic; Satisfiability Modulo Theories; Software Verification; Formal Methods.

1 Introduction

Over the years, computer manufacturers have experimented with different machine representations for real numbers [1]. The two basic ways to encode a real number are the fixed-point representation, usually found in embedded microprocessors and microcontrollers [2], and the floating-point representation, in particular, the IEEE floating-point standard (IEEE 754-2008), which has been adopted by many processors [3, 4].

Each encoding can represent a range of real numbers depending on the word-length and how the bits are distributed. A fixed-point representation of a number consists of an integer component, a fractional component and a bit for the sign, while the floating-point representation consists of an exponent component, a mantissa component and a bit for the sign. Numbers represented using a floating-point encoding have a much higher dynamic range than the fixed-point one (e.g, a `float` in C has 24 bits of accuracy, but can have values up to 2^{127}), while numbers represented using a fixed-point representation can have a greater precision than floating-point, but less dynamic range [5]. Furthermore, the IEEE floating-point standard contains definition that have no direct equivalent in a fixed-point encoding, e.g, two infinities ($+\infty$ and $-\infty$) and for signalling and

quiet `NaN`s (**N**ot **a** **N**umber, used to represent an undefined or unrepresentable value), denormal numbers, rounding modes, etc.

In this paper, we present ESBMC, a bounded model checker that uses Satisfiability Modulo Theories (SMT) solvers to verify single- and multi-threaded C/C++ code [6, 7]. The tool is able to encode the programs using either fixed-point arithmetics (using bitvectors) or floating-point arithmetic (using the SMT theory of floating-point numbers [8]). Initially, ESBMC was only able to encode `float`, `double` and `long double` using a fixed-point encoding (used in a wide range of applications in the verification of digital filters [9, 10] and controllers [11, 12]); the lack of a proper floating-point encoding, however, meant that ESBMC was not able to find an entire class of bugs, such as the one shown in Figure 1.

```
1 int main()
2 {
3   float x;
4   float y = x;
5   assert(x==y);
6   return 0;
7 }
```

Figure 1: Simple floating-point program with a bug.

The program shown in Figure 1 will never fail if verified with a fixed-point encoding. However, when using a floating-point encoding, `x` can be `NaN` and comparing `NaN`s, even with themselves, is always false [3]. In this scenario, the assertion in line 5 does not hold.

Support for verifying programs that rely on floating-point arithmetic is an important contribution to the software verification community, as it helps demonstrate the applicability of SMT-based verification to real-world systems.

The main original contributions of this paper are:

- We describe the verification process in ESBMC from the C program to the SMT formula encoding, including the solvers that support floating-point arithmetic, special cases when encoding the program, unused operators from the SMT standard and an illustrative example (Section 3).
- We demonstrate that our floating-point encoding based on MathSAT is able to outperform not only ESBMC with Z3, but also all the other approaches that participated in the most recent round of SV-COMP [13]. In particular, ESBMC/MathSAT is able to verify 169 benchmarks in 9977.4 s, while ESBMC/Z3 verifies 127 in 44992.7 s. ESBMC was the most efficient verifier for the floating-point subcategory in SV-COMP 2017, with 308 scores, followed by Ceagle [14] (298 scores), and CBMC [15] (264 scores) (Section 4).

2 The Efficient SMT-Based Context-Bounded Model Checker (ESBMC)

In this section, we present ESBMC, an open source, permissively licensed (Apache 2), cross platform bounded model checker for C and C++ programs. ESBMC was developed to perform bounded model checks on both sequential and concurrent programs using a range of SMT solvers, and has a proven track of bug finding in real-world applications [6,7,16]. The tool also implements a technique to prove the correctness of (some) unbounded programs: the k-induction algorithm; this approach has been applied to a large number of benchmarks and has produced more correct results than similar competing tools [17]. Fig. 2 shows the tool architecture. Rounded white rectangles represent input and output; squared gray rectangles represent the verification steps.

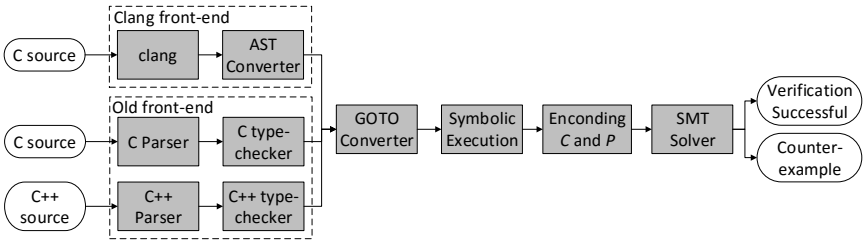

Figure 2: ESBMC architecture for floating-point verification.

ESBMC has two alternative front-ends to parse the input program and generate an an Abstract Syntax Tree (AST). There is the legacy CBMC-based front-end that supports both C and C++, and a new clang-based front-end that currently only supports C. The data types are created in the front-end when parsing the code, setting variable types to either fixed-point or floating-point for `float`, `double` and `long double`, depending on the options set by the user. Bitvector representations of constants are also created by the frontend, according to the fixed-point or floating-point semantics. The bitvector representation [7] of other data types (e.g., `int`, `char`) were not changed by the work described in this paper.

Regardless of the chosen front-end, the output is an AST that will be used by the GOTO converter to generate a GOTO program, which has simplified control flow and is suitable for bounded unwinding. The next step is the symbolic execution, when the GOTO program is executed (unrolling loops up to the bound k) and converted to Static Single Assignments (SSA) [18] form. During the symbolic execution, ESBMC aggressively tries to simplify the program; it propagates all constants and solves any assertions that can be statically determined. This is an important step for the verification; ESBMC can fully verify programs without calling a solver, if the inputs are deterministic.

The SSA expressions are then encoded using the chosen SMT solver; ultimately we are attempting to determine whether a formula, which is the disjunction of all possible errors, can be satisfied. If the SMT formula is shown to be satisfiable, a counterexample is presented; if the formula is found to be unsatisfiable, there are no errors up to the unwinding bound k, and this result is presented. ESBMC supports 5 SMT solvers: Boolector (default) [19], Z3 [20], MathSAT [21], Yices [22] and CVC4 [23]. In order to support and maintain this number of solvers, an SMT layer was developed, in such way that the support for new solvers, or new features like the floating-point support, only requires the implementation of the specific API calls for each solver.

It is in this layer that most of our contribution is concentrated. We implemented the new floating-point API in the SMT layer and the corresponding function calls for Z3 and MathSAT. The remaining solvers do not support floating-point arithmetic so ESBMC aborts the verification if an user tries to use this functionality with them.

3 Floating-point SMT Encoding

Here we describe our main contribution, the bit-precise encoding for ANSI-C programs using the SMT theory of floating-point. The SMT theory of floating-point covers almost all the operations performed at program level so the conversion is one-to-one and follows the encoding as described by Cordeiro et al. [7]. Given that, we focus on the limitations of the SMT theory of floating-point (casts to boolean types in Section 3.1 and the equality operator in Section 3.2) and how they were circumvented. In this section we also show operators from the SMT theory that are not being used in our implementation in Section 3.3 and an illustrative example of verification using the SMT theory of floating-point in Section 3.4; in this section we show the encoding and the counterexample generated by ESBMC, and the models generated by the solvers.

The SMT floating-point theory is an addition to the SMT standard, first proposed in 2010 by Rümmer and Wahl [8]. The current version of the theory largely follows the IEEE standard 754-2008 [3] and formalises the floating-point arithmetic, positive and negative infinities and zeroes, `NaN`s, comparison and arithmetic operators, and five rounding modes: round nearest with ties choosing the even value, round nearest with ties choosing away from zero, round towards positive infinity, round towards negative infinity and round towards zero. There are, however, some functionalities from the IEEE standard that are not yet supported by the SMT theory as described by Brain et al. [24].

Encoding programs using the SMT floating-point theory has several advantages over a fixed-point encoding, but the main one is the correct modeling of ANSI-C/C++ programs that use IEEE floating-point arithmetic. We created models for most of the current C11 standard functions [25]; floating-point exception handling, however, is not yet supported.

Currently, only two SMT solvers support the SMT floating-point theory: Z3 [20] and MathSAT [21] and ESBMC implements the floating-point encoding

for both. In terms of the support from the solvers, Z3 implements all operators, while MathSAT implements all but two: `fp.rem` (remainder operator) and `fp.fma` (fused multiply-add).

Both solvers offer two (non-standard) functions to convert floating-point numbers to and from bitvectors: `fp_as_ieeebv` and `fp_from_ieeebv`, respectively. These functions can be used to circumvent any lack of operators, and only require the user to write the missing operators.

3.1 Casts to boolean

The SMT standard defines conversion operations to and from signed and unsigned bitvectors, reals, integers and other floating-point types, but does not define a conversion operation for boolean types. ESBMC, however, generates these operations, as shown by the program in Figure 3. The program in Figure 3

```
1  int main(){
2    _Bool c;
3    double b = 0.0f;
4
5    b = c;
6    assert(b != 0.0f);
7
8    c = b;
9    assert(c != 0);
10 }
```

Figure 3: Program to demonstrate the casts to and from boolean generated by ESBMC.

forces ESBMC to generate two casts: one from boolean to double in line 5 and one from double to boolean in line 8. Figure 4a and Figure 4b present the SMT formula generated by these lines, respectively. When casting from booleans to floating-point numbers (Figure 4a), an `ite` operator is used, such that the result of the cast is 1.0 if the boolean is true; otherwise the result is 0.0. When casting from floating-point numbers to booleans (Figure 4b), we encode as a conditional assignment: the result of the cast is true when the floating is not 0.0; otherwise the result is false.

3.2 The `fp.eq` operator

Figure 4b also shows the second special cases when encoding ANSI-C programs. When encoding the program, both assignments and comparison operations are encoded using equalities. This must be changed, however, as the SMT standard defines a custom operator for floating-point equalities, `fp.eq` operator:

```
(assert (= (ite |main::c|
        (fp #b0 #b01111111111 #x0000000000000)
        (fp #b0 #b00000000000 #x0000000000000))
   |main::b|))
```

(a) SMT generated when casting from boolean to floating-point.

```
(assert (= (not (fp.eq |main::b|
        (fp #b0 #b00000000000 #x0000000000000)))
   |main::c|))
```

(b) SMT generated when casting from floating-point to boolean.

Figure 4: SMT formula generated by ESBMC to encode the casts to and from boolean types in Figure 3.

```
:note
"(fp.eq x y) evaluates to true if x evaluates to -zero and y
to +zero, or vice versa. fp.eq and all the other comparison op-
erators evaluate to false if one of their arguments is NaN."
```

In this case, the operator is defined to handle the special symbols from the IEEE floating-point standard, in particular, `NaN`s. It would not be correct to use the ordinary operator equality for comparison; it should only be used for assignments, while `fp.eq` is used for comparing floating-point numbers.

3.3 Unused operators from the SMT standard

When implementing the floating-point encoding, we did not use four operators defined by the SMT standard: `fp.max`, `fp.min`, `fp.rem` and `fp.isSubnormal`, instead we reimplemented them for enhanced perfomance:

1. `fp.max`: returns the larger of two floating-point numbers; equivalent to the `fmax`, `fmaxf`, `fmaxl` functions. Our model of the functions is shown in Figure 5.
2. `fp.min`: returns the smaller of two floating-point numbers; equivalent to the `fmin`, `fminf`, `fminl` functions. Our model of the functions is shown in Figure 6.
3. `fp.rem`: returns the floating-point remainder of the division operation `x/y`; equivalent to the `fmod`, `fmodf`, `fmodl` functions. Our model of the functions is shown in Figure 7.

```
double fmax(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x)) return y;
  if(isnan(y)) return x;

  return (x > y ? x : y);
}
```

Figure 5: Model for fmax.

```
double fmin(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x) || isnan(y)) {
    if(isnan(x))
      return y;
    return x;
  }

  return (x < y ? x : y);
}
```

Figure 6: Model for fmin.

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0))
    return signbit(x) ? -0.0 : +0.0;

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Figure 7: Model for fmod.

4. `fp.isSubnormal`: checks if a number is subnormal, i.e., a non-zero floating-point number with magnitude less than the magnitude of that format's smallest normal number. A subnormal number does not use the full precision available to normal numbers of the same format [3]. We could not find any user case for it when modelling C11 standard functions.

3.4 Illustrative Example

As an illustrative example of the SMT encoding using the floating-point arithmetic, Figure 8 shows the full SMT formula generated by ESBMC[1] for the program in Figure 1, as printed by Z3.

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Figure 8: SMT formula generated by ESBMC for the program shown in Figure 1.

[1] ESBMC actually generates a slightly different SMT formula, which includes all the symbols used for the memory model. The variable names are also more elaborate as the generated SSA has to reflect different valuations of the variable: the variable storage in memory, the thread to which the variable is associated, the specific thread interleaving the variable is related to, and the valuation of the variable at different points in the program. Each valuation is represented by a symbol (`@`, `!`, `&` and `#`) and an index. They were omitted to make the formula easier to read.

The SMT formula contains all the symbol declaration (`main::x` and `main::y`), nondeteministic symbols (`nondetsymex::nondet0`) and a boolean variable (`execution_statet::\\guard_exec`), that evaluates to true if there is a property violation in the program. The pervasive occurrence of `FloatingPoint 8 24` derives from the exponent and mantissa lengths of single precision `float`s.

Both SMT solvers correctly find a failure model for the program; Z3 produces:

```
sat
(model
  (define-fun |main::x| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |main::y| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |execution_statet::\\\\guard_exec| () Bool
    true)
)
```

and MathSAT produces:

```
sat
( (|main::x| (_ NaN 8 24))
  (|main::y| (_ NaN 8 24))
  (|nondet_symex::nondet0| (_ NaN 8 24))
  (|execution_statet::\\guard_exec| true) )
```

This is the expected result from the verification of the program in Figure 1; the program violates the assertion if `x` (and consequently `y`) is `NaN`. This happens because `x` is left uninitialized.

The model generated by both solvers is converted back to SSA by ESBMC, that prints the assignments that lead to a property violation[2]. Figure 9 shows the counterexample presented by ESBMC when verifying the program in Figure 1, using the floating-point arithmetic to encode the program. This is the counterexample generated when verifying the program with MathSAT; the counterexample generated by Z3 presents a positive `NaN`, but it is otherwise the same (both solvers are correct and either a positive or a negative `NaN` will lead to a property violation in the program). ESBMC also presents the IEEE bitvector representation of the values assigned to the variables, whenever possible.

[2] In comparison, no model is generated by the solver when verified using the fixed-point arithmetic.

```
Counterexample:

State 1 file main3.c line 3 function main thread 0
main
------------------------------------------------------
  main3::main::1::x=-NaN (11111111100000000000000000000001)

State 2 file main3.c line 4 function main thread 0
main
------------------------------------------------------
  main3::main::2::y=-NaN (11111111100000000000000000000001)

State 3 file main3.c line 5 function main thread 0
main
------------------------------------------------------
Violated property:
  file main3.c line 5 function main
  assertion
  (_Bool)(x == y)

VERIFICATION FAILED
```

Figure 9: Counterexample generated by ESBMC when verifying the program in Figure 1.

4 Experimental Evaluation

This section is split into three parts. The description of benchmarks and setup is described in Section 4.1, while Section 4.2 describes the experimental objectives. In Section 4.3, we evaluate our encoding using two state-of-the-art SMT solvers (MathSAT and Z3) and compare our best approach to other verifiers that support floating-point arithmetic in Section 4.4.

4.1 Description of Benchmarks and Setup

We evaluate our approach using a set of verification tasks in the *ReachSafety-Floats* sub-category of SV-COMP, which contains programs using floating-point arithmetic [13]. As defined by the competition rules, we assume a 32-bit architecture and, for all benchmarks, we check the following property as specified by the SV-COMP rules:

```
CHECK( init(main()), LTL(G, !call(__VERIFIER_error())) )
```

which means that from the `main()` function, we check the reachability of the function `__VERIFIER_error()` through any possible program execution. If there is a path from the program start to `__VERIFIER_error()`, the program contains a bug.

All experiments were conducted on a computer with an Intel Core i7-2600 running at 3.40GHz and 24GB of RAM under Fedora 25 64-bit. For each benchmark, we set time and memory limits of 900 seconds (15 minutes) and 16GB respectively. We provide a package with the latest version of ESBMC, all the benchmarks and the scripts to run the experiments at `http://esbmc.org/benchmarks/pack-sbmf2017.tar.gz`.

4.2 Objectives

Using the SV-COMP floating-point benchmarks given in Section 4.1, our experimental evaluation aims to answer two research questions:

RQ1 **(performance)** does our encoding generate verifications conditions that can be checked by state-of-the-art SMT solvers in a reasonable amount of time?

RQ2 **(sanity)** are the verification results sound and can their reproducibility be confirmed outside of our verifier?

4.3 Solver Performance Comparisons

	ESBMC (MathSAT v5.3.14)	ESBMC (Z3 v4.5.0)
Correct true	**139**	111
Correct false	**30**	16
Timeout	**3**	45
Total time (s)	**9977.40**	44992.76

Table 1: Comparative results of ESBMC using MathSAT v5.3.14 and Z3 v4.5.0.

Table 1 compares the results of ESBMC using both solvers on 172 benchmarks from SV-COMP'17, using a fixed unwind approach. Here, *Correct true* is the number of correct positive results (i.e., the tool reports SAFE correctly), *Correct false* is the number of correct negative results (i.e., the tool reports UNSAFE correctly), *Timeout* represents the number of time-outs (i.e., the tool was aborted after 900 seconds) and *Total time* is the total verification time, in seconds. Bold numbers represent better results. There is no case where ESBMC reports an incorrect result or exhausts the memory, so we are omitting them from the table.

When verifying the programs, ESBMC is able to statically verify 76 out of the 172 benchmarks (44.18%). This is due to the fact that the these programs are deterministic and, as described in Section 2, ESBMC is able to verify the program without calling a solver.

For the programs that ESBMC requires a solver for the verification, the verification time for both solvers is considerably longer when arrays are

present, in comparison to array-free programs. Given the set of benchmarks, ESBMC/MathSAT is able to solve all but three (within the time limit), while ESBMC/Z3 times-out for most array programs with an increased verification time for the others.

ESBMC/Z3 also fails to verify the same 3 benchmarks as ESBMC/MathSAT. The 3 programs[3] were created by Delmas et al. [26] and try to calculate a sine over a range of nondeteministic values, using an interpolation table. These programs assume a range of nondeterministic input and contains arrays, requiring a great deal of time to find the solution.

We provide a table that includes the number of variables, the number of clauses as well as the number of conflicts on the package previously mentioned in Section 4.1. However, we are unable to draw conclusions based on the provided numbers since we do not identify any pattern. For instance, one of the benchmarks that cannot be solved by MathSAT, `sin_interpolated_bigrange_tight_true-unreach-call.c` generates 545667 variables and 2240257 clauses, while 794852 variables and 3263282 clauses are generated when verifying `sin_interpolated_negation_true-unreach-call.c` and the latter can be verified in 61.3 seconds.

We can, however, compare these numbers, generated by both solvers. It is clear that MathSAT is more aggressively simplifying the program before bit-blasting. The total numbers are: Z3 generates 1.4×10^{10} variables and 1.4×10^{10} clauses in 44992.76 seconds (12.5 hours), while MathSAT generated 1.21×10^7 variables 4.73×10^7 clauses in 9977.40 seconds (2.8 hours). In terms of total numbers of conflicts, MathSAT generates $1.7\times$ more conflicts than Z3. However, Z3 was not able to finish the verification of 26.1% of the benchmarks, so this number is at best an approximation of the real value (this is not true for the number of variables and clauses, since they do not change during the execution of the DPLL algorithm [27]).

4.4 Comparison to other Software Verifiers

ESBMC with MathSAT greatly outperforms Z3 when verifying the competition's benchmarks and was the solver we used for the competition. Figure 10 shows the results of all verifiers on the *ReachSafety-Floats* sub-category of SV-COMP'17.

Figure 10 relates each tool score (y-axis) to the time spent during verification, in seconds (x-axis). Note that verifiers which actually give incorrect results can accumulate negative scores [13]. Using the fixed unwind approach, ESBMC was able to verify all but the 3 benchmarks previously mentioned, with a final score of 308 out of 316, in 5200 seconds, followed by Ceagle [14], with final score of 298 int 15000 seconds, and CBMC [15], with a final score of 264 in 3000 seconds. ESBMC also competed with other approaches in the competition, *ESBMC-falsi*, an incremental approach focused on finding bugs, *ESBMC-incr*, an incremental approach that provides a successful answer when it unrolled all loops, and

[3] `sin_interpolated_index_true-unreach-call.c`, `sin_interpolated_bigrange_loose_true-unreach-call.c` and `sin_interpolated_bigrange_tight_true-unreach-call.c`

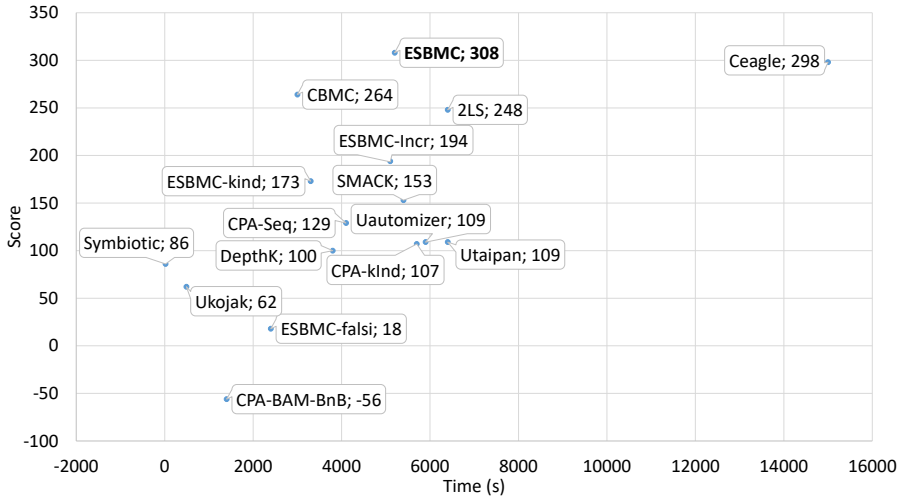

Figure 10: ESBMC architecture for floating-point verification.

ESBMC-kind, that tries to find bugs and prove correctness using induction [17]; these 3 approaches use Z3 as it performs better in other categories. The results from SV-COMP'17 are on par with the results presented in Section 4.3, where MathSAT outperforms Z3 when verifying programs with floating-point arithmetic.

These results allow us to answer both the research questions proposed in Section 4.2. The first inquires after the performance of our solver. Under the limits imposed by SV-COMP'17 (15 minutes and 16GB of RAM), ESBMC with MathSAT or Z3 is able to verify 98.2% and 73.8% of the benchmarks, respectively. The verification time is almost half of the presented in our results, due to the fact that SV-COMP has faster processors [13].

The second research question enquiries about the soundness and reproducibility of the results. The benchmarks from SV-COMP are thoroughly tested by all the verifiers, months before the actual competition, to ensure that all the verdicts are correct. ESBMC was able to encode all benchmarks and no wrong result was provided by our tool. However, SV-COMP still lacks the ability to automatically reproduce the counterexamples produced by verifiers in the *ReachSafety-Floats* sub-category, mainly because of the availability of witness checkers; these currently do not handle floating-point arithmetic [13].

5 Related Work

SMT solvers are an improving technology, being able to reason about ever growing formulas. These constant improvements feed the creation of a number of SMT-based software verification tools to the extent that they are already being

applied in industry (e.g., Static Driver Verifier [28]). Here, we present other tools that bridge the gap between a C/C++ program and the SMT solver.

Wang et al. [14] describe Ceagle, an automated verification tool for C programs. The tool applies 4 different approaches when verifying a program: (1) a bounded model checker with a fixed unwind approach that uses SMT to check for satisfiability; (2) a predicate lazy abstraction engine which verifies the program with a predicate-based abstract model and uses CEGAR to refine spurious counterexamples; (3) a structural abstraction engine which tries to reason about the program behaviour based on the program structure; and (4) an execution engine, which is executed when all parameters are deterministic. The tool competed in SV-COMP'17 and was ranked 2nd, if we consider only the *ReachSafety-Floats* sub-category. Ceagle was the only tool that was able to verify the 3 programs that ESBMC/MathSAT could not handle under the 15 minutes constraint; it was able to verify each one of them in less than 10 seconds.

Clarke et al. [15] describe CBMC, a C/C++ SAT/SMT bounded model checker. ESBMC originated as a fork of this tool with an improved SMT backend and support for the verification of concurrent programs using an explicit interleaving approach. CBMC uses SAT solvers as their main engine, but offers support for the generation of an SMT formula for an external SMT solver. ESBMC supports SMT solvers directly, through their APIs, along with the option to output SMT formulae. CBMC also competed in SV-COMP'17 with a fixed unwind approach, and was ranked 3rd in the *ReachSafety-Floats* sub-category.

Brain et al. [29] describe 2LS, C/C++ SAT/SMT bounded model checker. 2LS is a tool developed using the CPROVER framework [15] and aims to combine a k-induction algorithm with abstract interpretation. As CBMC, 2LS uses SAT solvers but instead of a fixed unwind approach, 2LS uses an incremental bounded model checking approach, where it first checks for property violations for a given bound, then tries to generate (and refine) invariants using abstract interpretation and then builds a proof using the k-induction algorithm. 2LS competed in SV-COMP'17 and was ranked 4th in the *ReachSafety-Floats* sub-category.

Compared to these tools, ESBMC is able to verify a program either using a fixed unwind approach (as Ceagle and CBMC) or an incremental BMC (as 2LS). Similar to Ceagle, ESBMC directly uses the solver API to encode the SMT formula, but ESBMC supports more SMT solvers than Ceagle (in particular, Ceagle only supports Z3).

Regarding the SMT solvers, after MathSAT and Z3, we expect the SMT solver CVC4 to support the floating-point theory shortly, as the code appears to be ready and waiting to be merged on its public repository. XSAT [30] is another solver that claims to be a "fast floating-point satisfiability solver": up to 700x faster than MathSAT and Z3 on the benchmarks from the International SMT competition. We were, however, unable to find the solver online to experiment with it.

6 Conclusions and Future Work

This paper presents a BMC approach to encoding C programs using SMT floating-point theory, evaluates the encoding using the SMT solvers that support this theory, and compares our approach with other existing floating-point verification tools.

The encoding was implemented in ESBMC, an SMT-based bounded model checker for C and C++ programs. ESBMC supports most of the current C11 standard functions and part of the floating-point environment behaviour; we currently support changing rounding modes, but floating-point exception handling is not yet supported.

We evaluated the results using two state-of-art SMT solvers, MathSAT and Z3, over a set of public benchmarks from the International Competition on Software Verification (SV-COMP) and the results show that, when using MathSAT, ESBMC is not only able to produce better results than Z3, but it is also able to produce better results than all other verifiers in SV-COMP.

For future work, we intend to create our own floating-point backend, so we are able to encode all the floating-point operations defined by the standard using bitvectors; this will allow us to use all available SMT solvers that support QF_BV when verifying programs with floating-point numbers.

Regarding the benchmarks, although we have reported a favourable assessment of ESBMC over a diverse set of floating-point benchmarks, this set of benchmarks is still of limited scope and ESBMC's performance needs to be assessed on a larger benchmark set in future.

References

1. Gerrity, G.W.: Computer representation of real numbers. IEEE Transactions on Computers **C-31**(8) (1982) 709–714
2. Frantz, G., Simar, R.: Comparing fixed- and floating-point DSPs. SPRY061, Texas Instruments (2004)
3. IEEE: IEEE standard for floating-point arithmetic. Technical report (Aug 2008)
4. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**(1) (1991) 5–48
5. Nikolić, Z., Nguyen, H.T., Frantz, G.: Design and implementation of numerical linear algebra algorithms on fixed point DSPs. EURASIP J. Adv. Sig. Proc. **2007**(1) (2007)
6. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE. (2011) 331–340
7. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering **38**(4) (2012) 957–974
8. Rümmer, P., Wahl, T.: An SMT-lib theory of binary floating-point arithmetic. In: SMT Workshop. (2010)
9. Ismail, H., Bessa, I., Cordeiro, L.C., Filho, E.B.d.L., Filho, J.E.C.: DSVerifier: A bounded model checking tool for digital systems. In: SPIN. Volume 9232 of LNCS. (2015) 126–131

10. Abreu, R.B., Gadelha, M.Y.R., Cordeiro, L.C., Filho, E.B.d.L., da Silva Jr., W.S.: Bounded model checking for fixed-point digital filters. Journal of the Brazilian Computer Society **22**(1) (2016) 1:1–1:20
11. Bessa, I., Ismail, H., Cordeiro, L.C., Filho, J.E.C.: Verification of fixed-point digital controllers using direct and delta forms realizations. Design Automation for Embedded Systems **20**(2) (2016) 95–126
12. Bessa, I., Ismail, H., Palhares, R., Cordeiro, L.C., Filho, J.E.C.: Formal non-fragile stability verification of digital control systems with uncertainty. IEEE Transactions on Computers **66**(3) (2017) 545–552
13. Beyer, D.: Software verification with validation of results (report on sv-comp 2017). In: TACAS. Volume 10206 of LNCS. (2017) 331–349
14. Wang, D., Zhang, C., Chen, G., Gu, M., Sun, J.G.: C code verification based on the extended labeled transition system model. In: D&P@MoDELS. (2016) 48–55
15. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Volume 2988 of LNCS., Springer (2004) 168–176
16. Ramalho, M., Freitas, M., Sousa, F., Marques, H., Cordeiro, L.C., Fischer, B.: SMT-Based Bounded Model Checking of C++ Programs. In: ECBS. (2013) 147–156
17. Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. STTT **19**(1) (2017) 97–114
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An Efficient Method of Computing Static Single Assignment Form. In: POPL. (1989) 25–35
19. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: TACAS. Volume 5505 of LNCS. (2009) 174–177
20. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. Volume 4963 of LNCS. (2008) 337–340
21. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: TACAS. Volume 7795 of LNCS. (2013) 93–107
22. Biere, A., Bloem, R., eds.: Yices 2.2. In Biere, A., Bloem, R., eds.: CAV. Volume 8559 of LNCS., Springer (2014)
23. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. Volume 6806 of LNCS. (2011) 171–177
24. Brain, M., Tinelli, C., Ruemmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: ARITH. (2015) 160–167
25. Smith, R.: Working Draft, Standard for Programming Language C++. (2016) [Online; accessed January-2017].
26. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: FMICS 2009. (2009) 53–69
27. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (1962) 394–397
28. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: CAV. Volume 6174 of LNCS. (2010) 119–122
29. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: SAS. (2015) 145–161
30. Fu, Z., Su, Z.: Xsat: A fast floating-point satisfiability solver. In: CAV. Volume 9780 of LNCS. (2016) 187–209