University of Utah
UNDERGRADUATE RESEARCH JOURNAL

EXTENDING SUPPORT FOR FLOATING-POINTS IN THE BOOGIE AND SMACK SOFTWARE VERIFIERS

Liam Machado (Advised by Zvonimir Rakamarić)

School of Computing

ABSTRACT

Software verification, which aims to prove critical properties about programs using rigorous formal methods based on logic, is an active area of research in the field of computer science. In particular, the verification of floating-points is a topic of paramount importance, given their ubiquity across a wide range of software, including video games, OS kernels, medical devices, and rockets. Several years ago, floating-point support was added to the Boogie verifier, a tool that is widely used by researchers in both industry and academia. In this thesis, I present the expansion of Boogie's original implementation to support rounding modes, an updated syntax for floating-point constants, and the fixing of various bugs, among other changes. I also present the addition of support for the math.h functions in the SMACK toolchain, which utilized Boogie's updated implementation in order to do so. Finally, I compare the performance of the updated implementation against other competitive verification tools on a comprehensive set of floating-point benchmarks.

CONTENTS

ABSTRACT ii

LIST OF FIGURES iv

LIST OF TABLES v

ACKNOWLEDGEMENTS vi

1 INTRODUCTION 1

2 BACKGROUND 3
- 2.1 Boogie 3
- 2.2 SMACK 3
- 2.3 Existing Floating-Point Support in Boogie 3

3 ENHANCING SUPPORT FOR FLOATING-POINTS 9
- 3.1 Correct Types Returned for Floating-Point Expressions 9
- 3.2 Valid Programs Passed to Corral 9
- 3.3 Redesign of Floating-Point Constants 10
  - 3.3.1 No Synonyms for Special Values 10
  - 3.3.2 Readable Mantissa and Exponent 10
  - 3.3.3 Unnormalized Significand 11
- 3.4 Updated Constant Syntax 12
- 3.5 Rounding Mode Support 12
  - 3.5.1 Using a Rounding Mode with ”builtin” Functions 13
- 3.6 Code Reorganization 14

4 APPLICATIONS 15
- 4.1 SMACK Support for math.h Functions 15
  - 4.1.1 Regressions 16
- 4.2 SMACK Support for Rounding Modes 18
- 4.3 O2Controller 18

5 EXPERIMENTS 19

6 RELATED WORK 20

7 CONCLUSIONS AND FUTURE WORK 21

LIST OF FIGURES

2.1 The SMACK Toolchain 3

LIST OF TABLES

2.1 Special Values 4
2.2 Built-in Operators 5
2.3 Operators Accessible with the "builtin" Attribute 5
2.4 Rounding Modes 6
2.5 Type Conversion Functions 7

4.1 math.h Functions Supported by SMACK 17

5.1 SV-COMP Benchmark Results 19

ACKNOWLEDGEMENTS

I would like to thank Dietrich Geisler for implementing the original floating-point support in Boogie. I would also like to thank Rustan Leino for designing the updated Boogie syntax for floating-point constants. This work was supported by funding from the Undergraduate Research Opportunities Program at the University of Utah and the National Science Foundation award CCF-1704715.

1 INTRODUCTION

Software verification, which aims to prove properties about programs using rigorous formal methods based on logic, is an active and critical area of research in the field of computer science. Because of the magnitude and importance of verifying that a program works as intended, a myriad of software tools have been developed for this task (see the results of the annual software verification competition SVCOMP for a list of such tools [35]). In the past several decades, researchers have invented a number of software verification techniques. More recently, techniques based on automated theorem provers have been shown to be particularly promising. There are several steps involved in performing verification using such techniques. The actual verification is typically performed by a *Satisfiability Modulo Theories* (SMT) solver, which takes as input the program expressed as a series of mathematical logic statements, and utilizes a variety of algorithms in order to solve for them. However, the issue remains of translating an input source file into a set of logic statements.

One tool that can be leveraged to bridge this gap is the Boogie *intermediate verification language* (IVL), an intermediate representation that lies between high-level programming languages and first-order logic format [3]. Verification tools including Corral [22] and Boogie [3] have been developed that can convert Boogie IVL programs into first-order logic statements, pass these statements to a built-in SMT solver such as Z3 [27], and output the verification result.

Another tool that aids in this process is the SMACK verification toolchain [31], which is capable of converting LLVM Intermediate Representation (IR) into Boogie IVL. LLVM [24] is a compiler infrastructure that includes a tool called Clang [8]. Clang translates source files written in high-level programming languages into a single universal intermediate representation and that has support for many popular languages including C, C++, C#, Java, and Python. By using the intermediary languages of Boogie and LLVM IR, a tool such as SMACK to translate between them, and a back-end verifier such as Corral to verify the resulting logic statements, users are able to verify the assertions in input programs.

Floating-point verification is a sub-field of software verification that has gained significant importance. Floating-point arithmetic appears in a wide variety of software applications, ranging from video games and device drivers to flight navigation systems and radiation therapy machines. However, there are relatively few verification tools that provide support for floating-point types. In 2016, Dietrich Geisler, a University of Utah alumni, added floating-point support to the Boogie verifier, basing the implementation on the SMT-LIB FloatingPoint theory [33]. This theory is supported by a variety of SMT solvers and is itself based off of the 2008 revision to the IEEE 754 standard [20]. The floating-point support was designed to provide a human-readable Boogie syntax while being easily translatable to SMT-LIB format. However, the initial implementation was incomplete and possessed a variety of bugs. Several parts of the SMT-LIB standard were missing in the implementation, including support for rounding modes. The implementation was incompatible with the Corral verifier, a tool which is dependent on Boogie, failing on simple floating-point regressions. Additionally, the syntax for floating-point constants was relatively complex and difficult to convert to and from standard decimal notation.

Dietrich also began adding support to SMACK for modeling the math.h functions from the C Standard Library, utilizing Boogie's floating-point implementation in order to do so.

However, this project was left mostly unfinished, and many of the implemented function models were incorrect.

My main contributions are:

- I added support for rounding modes to Boogie's existing floating-point implementation
- With the help of Rustan Leino, I devised and implemented a novel syntax for floating-point literals in Boogie
- Building on the initial work done by Dietrich, I added support in SMACK for verifying the math.h functions from the C Standard Library
- All of the source code that I wrote was peer-reviewed, tested, and merged by the Boogie and SMACK developers into their respective main trunks.

In this thesis, I begin by describing several tools referenced throughout subsequent chapters. Next, I present the original implementation of floating-points in Boogie. I follow this by describing the issues present in the original implementation, as well as how the updated implementation resolved each of these issues. I then detail how I utilized Boogie's updated implementation in order to finish adding support for the math.h functions in SMACK. Next, I present the experimental results of making these fixes. I did this by leveraging SMACK and running it against a comprehensive set of floating-point benchmarks obtained from the Software Verification Competition (SV-COMP) 2017 [35]. I then give an overview of the related work that has been done on floating-point verification, followed by my conclusions.

2 BACKGROUND

2.1 Boogie

Boogie refers to a procedural language designed as an intermediary form between high-level programming languages and first-order logic format. It is a platform that may be used to build verifiers for programs written in other languages [3]. Boogie also refers to a verification tool that accepts Boogie IVL code as input. The tool generates a set of weakest preconditions for the program [1] and passes the result to an SMT solver. Currently, Boogie has support for the Z3 and CVC4 [11] solver tools.

2.2 SMACK

The SMACK verification toolchain is capable of transforming LLVM Intermediate Representation (IR) [24] into Boogie code. LLVM IR is a low-level, platform independent instruction set, serving as a universal intermediate representation of programs written in higher-level languages. SMACK is also an end-to-end toolchain for software verification, providing a complete environment for users to analyze the assertions in input programs. Figure 2.1 depicts a visualization of this process. First, SMACK includes Clang [8], a sub-project of LLVM that translates input source files into LLVM IR. After the LLVM IR is run through several optimization passes, SMACK translates it into Boogie code. The Boogie code is then passed to a back-end verifier included with SMACK, which is responsible for computing the final verification result. SMACK comes packaged with four back-end verifiers: Corral, Boogie, Duality [13], and Symbooglix [23].

2.3 Existing Floating-Point Support in Boogie

In this section, I describe the features included in the original floating-point support added to Boogie in 2016, which was primarily developed by Dietrich Geisler. For more details regarding the development of this implementation, refer to previous work [29].

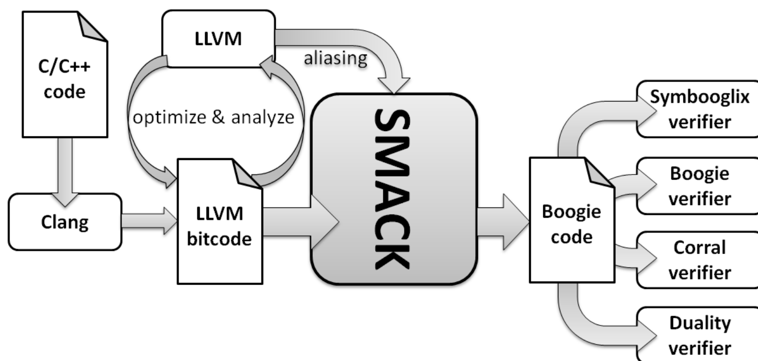

Figure 2.1: The SMACK Toolchain

Table 2.1: Special Values

Value	Declaration
+infinity	`0+oo[sigSize]e[expSize]`
-infinity	`0-oo[sigSize]e[expSize]`
NaN	`0NaN[sigSize]e[expSize]`

Type declarations A floating-point type with `sigSize` significand bits and `expSize` exponent bits is defined using the syntax `float[sigSize+1]e[expSize]` (the `+1` signifies an implicit bit in front of the decimal place). Both `sigSize` and `expSize` must be greater than 1. For example, a floating-point variable with 52 significand bits and 11 exponent bits may be defined like so:

```
var t : float53e11;
```

Constant declarations A floating-point constant is defined using the syntax `(-)[sig]e[exp]f[sigSize]e[expSize]`. In the above syntax, `sig` is the significand value, `exp` is the exponent value, `sigSize` is the number of significand bits, and `expSize` is the number of exponent bits. Finally, the presence of a `-` sign signifies a negative value.

For example, the constant -17.75 is defined using the following syntax:

```
var t : float11e5;
t := -112e19f11e5;
```

NaN and ±infinity may be defined using the above syntax. For example, +infinity may be defined in the following way:

```
var t : float11e5;
t := 0e31f11e5;
```

Alternatively, these special values may be defined using the syntax shown in Table 2.1.

Built-in operators There are several floating-point operators that are built into Boogie's syntax. These operators, as well as their corresponding syntax, are shown in Table 2.2. All arguments must share the same significand size and exponent size. If the return value is a float type, it is guaranteed to have the same significand size and exponent size as the arguments.

Accessing operators with the "builtin" attribute The "builtin" attribute enables access to operators that are not built into Boogie. These operators are listed in Table 2.3. As was the case with the built-in operators, all arguments must share the same significand size and exponent size. If the return value is a float type, it is guaranteed to have the same significand size and exponent size as the arguments.

These functions are accessed by creating a new function with 'function_name' that takes in type arguments 'arg_types' and returns a value of type 'return_type', with the syntax below:

Table 2.2: Built-in Operators

Operator	Built-in syntax	Return type
fp.neg(x)	-x	float
fp.add(x, y)	x + y	float
fp.sub(x, y)	x - y	float
fp.mul(x, y)	x * y	float
fp.div(x, y)	x / y	float
fp.leq(x, y)	x <= y	boolean
fp.lt(x, y)	x <y	boolean
fp.geq(x, y)	x >= y	boolean
fp.gt(x, y)	x >y	boolean

Table 2.3: Operators Accessible with the "builtin" Attribute

Operator	Return type
fp.abs(x)	float
fp.fma(x, y, z)	float
fp.sqrt(x)	float
fp.rem(x, y)	float
fp.roundToIntegral(x)	float
fp.min(x, y)	float
fp.max(x, y)	float
fp.isNormal(x)	boolean
fp.isSubnormal(x)	boolean
fp.isZero(x)	boolean
fp.isInfinite(x)	boolean
fp.isNaN(x)	boolean
fp.isNegative(x)	boolean
fp.isPositive(x)	boolean

```
function {:builtin "fp.op"} 'function_name'(args) returns (return_type);
```

Here is an example showing how to use the `fp.abs` function:

```
type float32 = float24e8;

function {:builtin "fp.abs"} abs(float32) returns (float32);

procedure foo() {
   var x : float32;
   var y : float32;
   x := -0e127f24e8; //x = -1
   y := abs(x);
   assert(y == 1);
}
```

Table 2.4: Rounding Modes

Rounding Mode	Acronym
roundNearestTiesToEven	RNE
roundNearestTiesToAway	RNA
roundTowardPositive	RTP
roundTowardNegative	RTN
roundTowardZero	RTZ

This same syntax may also be used for the built-in operators. For example, the following code accesses the "fp.lt" operator via the "builtin" attribute:

```
type float32 = float24e8;

function {:builtin "fp.lt"} less_than(float32, float32) returns (bool);

procedure foo() {
  var x : float32;
  var y : float32;
  x := -0e127f24e8; //x = -1
  y := 0e127f24e8; //y = 1
  assert(less_than(x, y));
}
```

Operators that Accept a Rounding Mode Some of the functions shown in Tables 2.2 and 2.3 require a rounding mode to be passed to it. This is done by passing it as an argument to the "builtin" attribute when using the syntax described in the previous section. The rounding modes supported by Boogie are shown in Table 2.4. For example, the following code shows how to declare the "fp.add" operator using the RTP rounding mode:

```
function {:builtin "fp.add RNE"} add_rne(float24e8, float24e8) returns
    (float24e8);
function {:builtin "fp.add RTP"} add_rtp(float24e8, float24e8) returns
    (float24e8);

procedure foo() {
  var x : float24e8;
  var y : float24e8;
  var z1 : float24e8;
  var z2 : float24e8;

  z1 := add_rne(x, y);
  z2 := add_rtp(x, y);

  assert(z1 <= z2);
  assert(z1 == x + y);
```

Table 2.5: Type Conversion Functions

Operator	Return Type
to_fp(float)	float
to_fp(real)	float
to_fp(bit_vec)	float
to_fp_unsigned(bit_vec)	float
fp.to_ubv(float)	bit_vec
fp.to_sbv(float)	bit_vec
fp.to_real(float)	real

```
}
```

Note that using operators with the built-in syntax defaults to the RNE rounding mode, as demonstrated by the second assertion.

Type Conversions The "builtin" attribute is also used to access several type conversion functions, which are listed in Table 2.5. The syntax for accessing these operators is slightly different than for operators described in previous sections. Specifically, `to_fp` is accessed by passing the string `(_ to_fp expSize sigSize)` to the "builtin" attribute. For example, the following code converts a 16-bit floating-point value to a 32-bit one:

```
type float16 = float11e5;
type float32 = float24e8;

function {:builtin "(_ to_fp 8 24) RTP"} float16_to_float32(float16)
    returns (float32);

procedure foo() {
   var f : float16;
   var g : float32;
   f := 0e15f11e5;
   g := float16_to_float32(f);
   assert(g == 0e127f24e8);
}
```

Additionally, `fp.to_ubv` and `fp.to_sbv` are accessed by passing the string `(_ op_name bvSize)` to the "builtin" attribute. Here is an example that converts a 32-bit floating-point value to a bit vector of size 32:

```
type float32 = float24e8;

function {:builtin "(_ fp.to_ubv 32) RTZ"} float32_to_bv32(float32) returns
    (bv32);

procedure foo() {
   var f : float32;
```

```
    var g : bv32;
    f := 2097152e127f24e8; // f = 1.25
    g := float32_to_bv32(f);
    assert(g == 1bv32);
  }
```

`fp.to_real` is accessed in the same way as operators in the previous sections.

3 ENHANCING SUPPORT FOR FLOATING-POINTS

I will now describe the changes made in the updated floating-point implementation in Boogie. These changes include the fixing of several significant bugs, revision of the syntax for floating-point constants, addition of rounding mode support, major code reorganization, and renaming of regressions.

3.1 Correct Types Returned for Floating-Point Expressions

In the original implementation, floating-point expressions were sometimes computed to have an incorrect return type. For example, Boogie threw an exception when provided the following program:

```
1       type Ref;
2       var Heap: HeapType;
3       type Field A B;
4       type HeapType = <A, B> [Ref, Field A B]B;
5
6       procedure mfl(one: float23e11, two: float23e11) returns () {
7          assert two == one;
8       }
```

The expression at line 8, `two == one`, will return a result of type `float23e11`, even though the `assert` expression expects a `Bool` value as an argument. I resolved this issue in a pull request, which can be viewed at [17].

3.2 Valid Programs Passed to Corral

Additionally, the original implementation resulted in errors within the Corral verifier. Corral is a verification tool that utilizes goal-directed symbolic search strategies in order to verify Boogie programs [10]. Additionally, Corral's implementation depends on Boogie's. Corral obtains intermediate versions of programs processed by Boogie, which it then further processes. However, the original floating-point implementation was resulting in Boogie generating syntactically incorrect intermediate programs. For example, consider the following Boogie program:

```
procedure foo(a: float24e8, b: float24e8) {
   a := 0e127f24e8; //a = 1
   b := 0e128f24e8; //b = 2
   assert (a + a == b);
}
```

Given the above program as input, Boogie will generate the following intermediate program:

```
procedure foo(a: float24e8, b: float24e8) {
   a := 0x2\^127; //a = 1
   b := 0x2\^128; //b = 2
```

```
    assert (a + a == b);
}
```

I resolved this issue in a pull request, which can be viewed at [9].

3.3 Redesign of Floating-Point Constants

The primary issue with the original floating-point syntax was that it was relatively difficult for a human to convert to and from standard decimal notation. It also possessed several features that were confusing to both new users and those unfamiliar with how floating-points are represented in memory, as specified by the IEEE 754-2008 revision. The redesign of floating-point constant syntax followed several guiding principles explicated by Rustan Leino at [32]. These guidelines are described below.

3.3.1 No Synonyms for Special Values

Under the original syntax, floating-point special values could be represented in multiple ways. For example, the floating-point constant +infinity (for a 23-bit significand and 8-bit exponent) may be represented as either `0+oo24e8` or `0e255f24e8`. This becomes potentially confusing for users who may not be able to tell whether they are writing a special value or a value that falls within the bounds of the valid floating-point range.

Under the new syntax, synonyms for special values are disallowed. ±infinity and NaN may only be written using the syntax listed in Table 2.1.

3.3.2 Readable Mantissa and Exponent

Consider representing the number 10 in Boogie, using a 23-bit significand and an 8-bit exponent. A human must use roughly the following process to perform the conversion:

$$10_{10} = 1010_2 = 1.01 * 2^3 = 1.01 * 2^{(2^{8-1}-1)-127} = 1.01 * 2^{130-127}$$

The final expression implies that the exponent value is 130 and the significand value is 101. Because the hidden-bit convention is used, the first 1 is implicit. Additionally, the significand must be 23 bits long. Therefore, the significand is actually $010_0000_0000_0000_0000_0000_2 = 2097152_{10}$. This corresponds to a Boogie constant string of `2097152e130f24e8`.

As illustrated above, converting from decimal to Boogie constant syntax is a relatively long and error-prone process. Converting the opposite direction is similar in difficulty. Ideally, the conversion process should be shorter, and the representation should be easier to read.

The new design makes several changes to the syntax to accomplish this. Firstly, it makes the exponent unbiased. The IEEE 754 standard specifies that a bias value is added to the exponent of the desired number to obtain the corresponding bit pattern. Specifically, for a floating-point value with a w-bit exponent, a bias of $2^{w-1} - 1$ is added to the exponent. This is done in order to provide a clean method of representing both positive and negative exponents. For example, to represent the value $0.25 = 1.0 * 2^{-2}$ using 8 bits for the exponent, a bias of $2^{8-1} - 1 = 127$ is added. Thus, 0.25 is represented within Boogie as `0e-2f24e8`.

Note that the value of the exponent is 125 instead of -2, since the bias of 127 is added to obtain the proper value.

However, while adding a bias is required to represent floating-points internally abiding by the IEEE-754 standard, it is completely unnecessary to do so at the user level. Users who are unfamiliar with this convention may mistakenly think that the value `0e125f24e8` is equivalent to $1.0 * 2^{125}$ instead of $1.0 * 2^{125-127}$. The new syntax thus changes the exponent to be unbiased, simplifying the syntax and making it easier to understand. With this change, the Boogie string to represent 10 changes from `2097152e130f24e8` to `2097152e3f24e8`.

The second change in the new syntax is the removal of the hidden bit convention. The IEEE 754 standard specifies that the first set bit in the significand is not explicitly included in its internal representation. In the original Boogie syntax, this convention was replicated. However, many users who are unfamiliar with this convention can mistakenly include it as part of the significand. The hidden-bit convention is unnecessary in user-level syntax, confuses users, and complicates the representation. In the new syntax, the first bit is included by prepending the significand with the string "`1.`".

Additionally, the significand is now written in hexadecimal notation instead of decimal. This accomplishes several things. Firstly, it simplifies conversion from binary; it is much easier to convert between binary and hexadecimal than it is to convert between binary and decimal. Secondly, it shortens the resulting constant string; using a higher base reduces the number of characters needed to write the significand. With the removal of the hidden-bit convention and a hexadecimal significand, the Boogie string to represent 10 changes from `2097152e3f24e8` to `1.200000e3f24e8`.

Finally, and most importantly, the significand is now written in a left-to-right manner instead of right-to-left. For example, consider the significand in the above example, which was originally treated as 010_0000_0000_0000_0000_0000. Left as is, this would be written in hexadecimal as 200000. However, to a human, it is counterintuitive to write this number starting from the right. When we write decimal numbers on paper, we start from the left and progress to the right. Thus, the significand should instead be treated as 0100_0000_0000_0000_0000_000. Therefore, the Boogie string to represent 10 changes from `1.200000e3f24e8` to `1.400000e3f24e8`.

Another important change it to allow the user to exclude unnecessary trailing zeroes if they choose. This further simplifies the Boogie string to `1.4e3f24e8`.

For consistency with the hexadecimal significand, the exponent is now in base 16 instead of base 2. To accommodate this change, the digit before the decimal point is also allowed to be any hexadecimal digit, not just a 1. This now results in a Boogie constant string of `A.0e0f24e8` to represent 10.

3.3.3 Unnormalized Significand

Finally, the new syntax allows the significand to be unnormalized. In other words, there can be arbitrarily many digits before and after the decimal point, provided that the floating-point value that the written constant is equivalent to can fit within the specified number of significand and exponent bits. Thus, 10 can be written in Boogie as `0xA0.0e-1f24e8`, `0xA.0e0f24e8`, `0x0.Ae1f24e8`, `0x0.0Ae2f24e8`, etc.

3.4 Updated Constant Syntax

In this section, I describe my implementation of Boogie's updated floating-point constant syntax, following the guidelines stated in previous sections.

A floating-point constant can be declared with the following syntax:

```
(-)0x[sig]e[exp]f[sigSize]e[expSize]
```

In the above line:

`sig` = hexdigit {hexdigit} '.' hexdigit {hexdigit}

`exp` = digit {digit}

`sigSize` = digit {digit}

`expSize` = digit {digit}

A floating-point number written using the above syntax is equivalent to the value $(-)(sig* 16^{exp})$. As long as the equivalent value fits in a floating-point variable with 'sigSize' significand bits and 'expSize' exponent bits, there are no restrictions on the values of 'sig' and 'exp'.

The significand must have trailing zeros such that the last nibble is fully included. For example, in order to represent a floating-point value that has a 24-bit significand with the bit pattern 1_0000_0000_0000_0000_0000_001 (including the hidden bit at the beginning) and an exponent of 0, it may be written as `0x1.000002e0f24e8`, but not `0x1.000001e0f24e8`.

As another example, we can assign the constant -2.25 to var 'name' by writing:

```
var name : float24e8;
name := -0x2.4e0f24e8;
```

It is disallowed to use synonyms of the special values listed in Table 2.1 (for example, `0x1.0e32f24e8` is equivalent to `0+oo24e8` but is a syntax error).

3.5 Rounding Mode Support

In the original implementation, Boogie had no support for reasoning about rounding modes. Tools that convert programs written in a high-level programming language to Boogie would be unable to do so if the program involved rounding modes. For example, consider the following snippet of C code:

```c
if (x == 1) {
   fesetround(FE_UPWARD);
} else {
   fesetround(FE_DOWNWARD);
}
```

```
float y = floor(3.5);
```

`fesetround` is a Standard Library function that allows the user to set the rounding mode used by the program. Without a way of explicitly representing and reasoning about rounding modes in Boogie, verification tools have no way of modeling the above code.

To resolve this issue, I added a rounding mode type to Boogie, accessible through the `rmode` keyword. It is capable of modeling the five rounding modes defined in the SMT-LIB FloatingPoint theory, listed in Table 2.4.

For example, the RoundTowardNegative rounding mode may be assigned to var 'mode' by writing:

```
var mode: rmode;
mode := RTN;
```

Or, alternatively, the full name may be used:

```
var mode: rmode;
mode := roundTowardNegative;
```

3.5.1 Using a Rounding Mode with "builtin" Functions

In the original implementation, SMT-LIB functions that used rounding modes needed to have them hard-coded in the function definition, as described in Section 4.3. If a user wanted to use a specific function with different rounding modes, they would have to include separate declarations for each rounding mode. However, with the addition of a rounding mode type, operators that accept rounding modes may now do so as an argument to the function itself, rather than as an argument to the "builtin" attribute. This is done by creating a new function with `function_name` that takes in arguments of types `arg_types`, along with an `rmode` variable, and returns a value of type `return_type`, using the syntax below:

```
function {:builtin "fp.op"} 'function_name'(rmode, args)
   returns(return_type);
```

For example, the following code demonstrates how to declare the "fp.sub" operation and use it with both the RNA and RTZ rounding modes:

```
function {:builtin "fp.sub"} sub(rmode, float24e8, float24e8) returns
   (float24e8);
.
.
.
.
var x : float24e8;
var y : float24e8;
var z : float24e8;

z := sub(RNA, x, y);
z := sub(RTZ, x, y);
```

Or, alternatively:

```
function {:builtin "fsub"} sub(rmode, float24e8, float24e8) returns
    (float24e8);
.
.
.
.
var mode : rmode;
var x : float24e8;
var y : float24e8;
var z : float24e8;

mode := RNA;
z := sub(mode, x, y);
mode := RTZ;
z := sub(mode, x, y);
```

Additionally, as explained in Section 4.3, operators that accept rounding modes may be declared to conform to a specific rounding mode that is specified as part of the "builtin" attribute.

3.6 Code Reorganization

The min, max, and rem floating-point functions originally could be accessed without the "builtin" attribute, which was inconsistent with the other SMT-LIB functions. I revised the implementation such that the "builtin" attribute is now required to access these functions.

Additionally, I made changes to the regressions included in the Boogie project that test the floating-point implementation. These regressions were originally named "float1.bpl", "float2.bpl", etc. Having uninformative names made it difficult to determine what feature each one was testing. Thus, I renamed these regressions to reflect their individual intended purpose.

I made both of the above changes in a pull request that can be viewed at [26].

4 APPLICATIONS

4.1 SMACK Support for math.h Functions

I used Boogie's floating-point implementation to add support in SMACK for verifying the C standard library math.h functions. This project was begun in 2017 by adding support for some of the math.h functions. However, most of the functions were either unimplemented or modeled incorrectly. To resolve these issues, I added models to SMACK for most of the remaining functions and rewrote many of the existing models. Each model includes explicit Boogie code that makes calls to relevant "builtin" functions. For example, SMACK implements the `fabs` function like so:

```
double fabs(double x) {
   double ret = __VERIFIER_nondet_double();
   __SMACK_code("@ := $abs.bvdouble(@);", ret, x);
   return ret;
}
```

Note how short the implementation is, due to there being an equivalent "builtin" absolute value function. Also note how the `__SMACK_code` function must be called in order to inject inline Boogie code into C programs. When SMACK verifies an input C program that includes a call to `fabs`, it will generate the following Boogie code to model the `fabs` function:

```
type bvfloat = float24e8;
function {:builtin "fp.abs"} $abs.bvfloat(i: bvfloat) returns (bvfloat);

var i: bvfloat; //the argument passed to fabs
var j: bvfloat; //the return value of fabs
.
.
.
.
.
j:= $abs.bvfloat(i);
```

Some functions require more involved implementations, due to not having an equivalent "builtin" function. For example, below is the code for modeling the `modf` function:

```
double modf(double x, double *iPart) {
   double fPart = __VERIFIER_nondet_double();
   if (__isinf(x)) {
      *iPart = x;
      fPart = 0.0;
   } else {
      *iPart = trunc(x);
      __SMACK_code("@ := $fsub.bvdouble($rmode, @, @);", fPart, x, *iPart);
   }
   if (__iszero(fPart)) {
      fPart = (__signbit(x)) ? -0.0 : 0.0;
```

```
    }
    return fPart;
}
```

Table 4.1 displays a list of the math.h functions currently supported by SMACK. The models may be viewed at [25].

4.1.1 Regressions

The vast majority of the math.h functions result in special behavior when passed ±0, ±infinity, or NaN. In order to help ensure that the models matched expected behavior for all cases, I wrote regressions in the SMACK project for each supported function. The regressions collectively total approximately 4000 lines of C code. For example, below is the regression that tests SMACK's model of the `fmod` function:

```
#include "smack.h"
#include <math.h>

// @expect verified
// @flag --bit-precise

int main(void) {
   double NaN = 0.0 / 0.0;
   double Inf = 1.0 / 0.0;
   double negInf = -1.0 / 0.0;

   double val = __VERIFIER_nondet_double();

   if (!__isnan(val) && !__isinf(val) && !__iszero(val)) {
      if (val > 0) {
         assert(fabs(val) == val);
      } else {
         assert(fabs(val) == -val);
      }
   }

   assert(fabs(0.0) == 0.0);
   assert(fabs(-0.0) == 0.0);
   int isNeg = __signbit(fabs(-0.0));
   assert(!isNeg);

   assert(fabs(Inf) == Inf);
   assert(fabs(negInf) == Inf);

   assert(__isnan(fabs(NaN)));

   return 0;
}
```

Table 4.1: math.h Functions Supported by SMACK

float fabsf(float)	double fmod(double, double)
float fdimf(float, float)	double modf(double, double*)
float roundf(float)	double copysign(double, double)
long lroundf(float)	double nan(const char*)
float rintf(float)	int isnormal(double)
float nearbyintf(float)	int issubnormal(double)
long lrintf(float)	int iszero(double)
float floorf(float)	int isinf(double)
float ceilf(float)	int isnan(double)
float truncf(float)	int isnegative(double)
float sqrtf(float)	int signbit(double)
float remainderf(float, float)	int fpclassify(double)
float fminf(float, float)	int finite(double)
float fmaxf(float, float)	long double fabsl(long double)
float fmodf(float, float)	long double fdiml(long double, long double)
float modff(float, float*)	long double roundl(long double)
float copysignf(float, float)	long lroundl(long double)
float nanf(const char*)	long double rintl(long double)
int isnormalf(float)	long double nearbyintl(long double)
int issubnormalf(float)	long lrintl(long double)
int iszerof(float)	long double floorl(long double)
int isinff(float)	long double ceill(long double)
int isnanf(float)	long double truncl(long double)
int isnegativef(float)	long double sqrtl(long double)
int signbitf(float)	long double remainderl(long double, long double)
int fpclassifyf(float)	long double fminl(long double, long double)
int finitef(float)	long double fmaxl(long double, long double)
double fabs(double)	long double fmodl(long double, long double)
double fdim(double, double)	long double modfl(long double, long double*)
double round(double)	long double copysignl(long double, long double)
long lround(double)	long double nanl(const char*)
double rint(double)	int isnormall(long double)
double nearbyint(double)	int issubnormall(long double)
long lrint(double)	int iszerol(long double)
double floor(double)	int isinfl(long double)
double ceil(double)	int isnanl(long double)
double trunc(double)	int isnegativel(long double)
double sqrt(double)	int signbitl(long double)
double remainder(double, double)	int fpclassifyl(long double)
double fmin(double, double)	int finitel(long double)
double fmax(double, double)	

4.2 SMACK Support for Rounding Modes

I added rounding mode support to SMACK by writing models for the `fegetround` and `fesetround` functions, which are C Standard Library functions that are part of the `fenv.h` header file. These functions allow for the reading and writing, respectively, of the rounding mode used by the program. The models for these functions may be found at [16].

4.3 O2Controller

I am currenctly using the updated floating-point support in Boogie and SMACK to verify O2Controller, a medical device that monitors oxygen flow used by patients, being developed by Dynasthetics [15], a Utah based company. O2Controller is implemented in C and contains over a thousand lines of heavy floating-point computations. I, along with Prof. Zvonimir Rakamarić, have collaborated with the developers of O2Controller in order to determine the most salient portions of the code to verify, add assertions accordingly, and verify the code using Boogie. Although this project is still a work in progress, Boogie has already shown promising results. It has caught several bugs in the program that Corral failed to catch within a reasonable time limit. For example, Boogie verified four key properties of this application in eleven minutes, while Corral spent more than three hours to verify just one of these properties.

5 EXPERIMENTS

In 2017, SMACK participated in the Software Verification Competition (SV-COMP), competing against a myriad of other verification tools. SV-COMP was designed as a way to spread awareness of the state of the art in verification technology, as well as establish an accepted set of benchmarks that researchers can use to compare the performance of tools. SV-COMP's benchmarks are divided into distinct categories that each test different aspects of programs, including loops, memory safety, and concurrency. ReachSafety-Floats is a category consisting of benchmarks that make heavy usage of floating-point operations. SMACK was using Corral as its back-end verifier during this time.

Table 5.1 shows the results of running a performance comparison between SMACK with Boogie, SMACK with Corral, Ceagle [37], and ESBMC [30] on the SV-COMP 2017 ReachSafety-Floats category benchmarks. The entry for the verifier, "SMACK w/ Boogie" corresponds to SMACK using Boogie's updated floating-point implementation. As the results illustrate, although SMACK obtained better results with Boogie's updated implementation, it was still outpaced by Ceagle and ESBMC, the winner and runner-up, respectively, of the ReachSafety-Floats category in SV-COMP 2017. These tools are likely much more optimized for float-point operations than SMACK is. Additionally, Boogie uses Z3 as its back-end SMT solver by default. These tools may be using SMT solvers that perform better than Z3 on floating-point expresions.

Table 5.1: SV-COMP Benchmark Results

Verifier	**# Solved**	**# Timeout**	**# Error**
SMACK w/ Boogie	92	70	10
SMACK w/ Corral	85	78	9
Ceagle	164	0	8
ESBMC	169	0	3

6 RELATED WORK

There currently exist a variety of verification tools that are capable of analyzing floating-point expressions. FPTaylor [34] is a tool designed to estimate the round-off error of floating-point calculations. A related tool, FPTuner [6], may be used to tune the precision of multiple floating-point expressions in order to achieve an error bound that is below some threshold. KLEE [4] is a symbolic execution engine that operates on LLVM IR and is able to execute statements involving floating-point variables.

ESBMC is a context-bounded model checker that placed first in the ReachSafety-Floats category of benchmarks. It has support for a variety of SMT solvers, including Z3 and MathSAT [7]. [18] shows that, when using MathSAT, ESBMC outperforms Z3 and approaches used by other tools in SV-COMP. Ceagle is another tool that obtained strong results in SV-COMP, placing second in the ReachSafety-Floats category.

CBMC [21], like ESBMC, is a bounded model checker for C/C++ programs. It comes packaged with a MiniSat-based solver that supports the SMT-LIB FloatingPoint theory. However, it also support external verifiers, including Z3, MathSAT, Yices 2 [14], and Boolector [28].

In SV-COMP 2019 [36], VeriAbs [12] placed first in the ReachSafety-Floats category. VeriAbs is a bounded model checker, but utilizes techniques including abstract acceleration and k-induction in order to scale for large loop iterations. Pinaka [5], which placed second in 2019, is a tool built on the CProver framework. It it capable of reasoning about rounding modes for floating-point operations.

Other verification tools that support floating-point functionality include Ultimate Automizer [19] and BLAST [2]. Many of the tools described in this chapter have competed in SV-COMP, and the results of this competition serve as a useful performance comparison.

7 CONCLUSIONS AND FUTURE WORK

I extended the existing floating-point support in the Boogie verification tool by making the syntax more readable, adding rounding mode support, changing the implementation to be consistent with the SMT-LIB floating-point standard, and fixing several bugs, including incompatibility with Corral. I used the updated implementation to finish adding support in SMACK for verifying the math.h functions from the C Standard Library. In addition, I used Boogie and SMACK's implementation to find and resolve bugs in O2Controller, a real-world application.

The performance of SMACK on floating-points still does not reach the level of the top-performing verification tools in SV-COMP, such as ESBMC and Ceagle. These tools are almost certainly taking advantage of algorithms and optimizations that SMACK is not. As future work, it would be useful to investigate what these approaches are and implement them into SMACK in order to improving its own performance.

Additionally, the work with O2Controller remains incomplete. So far, key portions of the code have been analyzed disjointly, but the entire codebase has not been analyzed in its entirety. I would like to fully verify O2Controller and see it be approved by the FDA for official usage.

Finally, although the regressions for evaluating the correctness of the math.h models in SMACK are comprehensive, they do not prove that the models match the specified behavior for these functions. As future work, I would like to utilize more advanced techniques, such as differential testing, to validate the accuracy of these models by comparing them with official implementations of the math.h functions.

BIBLIOGRAPHY

[1] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, pages 173–193, 2005.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. In *International Journal on Software Tools for Technology Transfer, vol. 9, no. 5-6*, pages 505–525, 2007.

[3] Boogie: An Intermediate Verification Language. `https://www.microsoft.com/en-us/research/project/boogie-an-Intermediate-Verification-language/`.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[5] Eti Chaudhary and Saurabh Joshi. Pinaka: Symbolic Execution meets Incremental Solving (Competition Contribution). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Proceedings of TACAS*, LNCS. Springer and Cham, 2019.

[6] W.F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous floating-point mixed-precision tuning. In *ACM SIGPLAN Notices, vol. 52*, pages 300–315, 2017.

[7] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[8] Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`.

[9] Fixed floating-point issues with corral. `https://github.com/boogie-org/boogie/pull/84`.

[10] Corral Program Verifier. `https://www.microsoft.com/en-us/research/project/q-program-verifier/`.

[11] Cvc4, the smt solver. `http://cvc4.cs.stanford.edu/web/`.

[12] Priyanka Darke, Sumanth Prabhu, Bharti Chimdyalwar, Avriti Chauhan, Shrawan Kumar, Animesh Basakchowdhury, R. Venkatesh, Advaita Datar, and Raveendra Kumar Medicherla. VeriAbs: Verification by Abstraction and Test Generation. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer, 2018.

[13] Duality. `https://www.microsoft.com/en-us/research/project/duality/`.

[14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, LNCS, pages 737–744. Springer and Cham, 2014.

[15] Dynasthetics. http://www.dynasthetics.com/.

[16] fenv.c. https://github.com/smackers/smack/blob/master/share/smack/lib/fenv.c.

[17] Fix #80: Return correct return type for floating-point expressions. https://github.com/boogie-org/boogie/pull/82.

[18] Mikhail Y. R. Gadelha, Lucas C. Cordeiro, and Denis A. Nicole. Encoding Floating-Point Numbers Using the SMT Theory in ESBMC: An Empirical Evaluation over the SV-COMP Benchmarks. In Simone Cavalheiro and José Fiadeiro, editors, *Formal Methods: Foundations and Applications*, LNCS. Springer, 2017.

[19] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate automizer with smtinterpol. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 641–643, 2013.

[20] 754-2008 - IEEE Standard for Floating-Point Arithmetic. https://ieeexplore.ieee.org/document/4610935.

[21] Daniel Kroening and Michael Tautschnig. CBMC: C bounded model checker (Competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of TACAS*, LNCS, pages 389–391. Springer, 2014.

[22] A. Lal, S Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification (CAV)*, pages 427–443, 2012.

[23] D. Liew, C. Cadar, and A. F. Donaldson. Symbooglix: A symbolic execution engine for boogie programs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 45–56, 2016.

[24] The LLVM compiler infrastructure project. http://llvm.org/.

[25] math.c. https://github.com/smackers/smack/blob/master/share/smack/lib/math.c.

[26] Modifications to floating-point support. https://github.com/boogie-org/boogie/pull/85.

[27] L. D. Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[28] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[29] Floating point support. https://github.com/boogie-org/boogie/pull/35.

[30] Felipe R. Monteiro, Mikhail Ramalho, Lucas Cordeiro, Bernd Fischer, Jeremy Morse, and Denis A. Nicole. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 888–891, 2018.

[31] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *International Conference on Computer Aided Verification (CAV)*, pages 106–113, 2014.

[32] Support for "+zero" and "-zero" special values. `https://github.com/boogie-org/boogie/pull/91`.

[33] SMT-LIB The Satisfiability Modulo Theories Library. `smtlib.cs.uiowa.edu/theories-FloatingPoint.shtml`.

[34] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM 2015: Formal Methods*, pages 532–550, 2015.

[35] Competition on Software Verification (SV-COMP). `https://sv-comp.sosy-lab.org/2017/`.

[36] Competition on Software Verification (SV-COMP). `https://sv-comp.sosy-lab.org/2019/`.

[37] Dexi Wang, Chao Zhang, Guang Chen, Ming Gu, and Jiaguang Sun. C code verification based on the extended labeled transition system model. In *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016)*, pages 48–55, 2016.