Predictive Capabilities of SMT Solvers

David Watkins Columbia University
New York, New York USA
May 13th, 2016
djw2146@columbia.edu

ABSTRACT

SMT solvers have, in recent years, undergone optimizations that allow them to be considered for use in commercial software. Usages for such SMT solvers include program verification, buffer overflow detection, bit-width prediction, and loop unrolling. Companies such as Microsoft have pioneered SMT research through their Z3 solver. In this paper I investigate the potential techniques for implementing these techniques as well as provide examples of potential applications of SMT solvers.

1. MOTIVATION

In coordination with Professor Stephen Edwards at Columbia University, the project *Hardware synthesis from a recursive functional language*? that compiles a subset of Haskell into a hardware description language has many potential sources of optimization. These optimizations would include loop unrolling, bit width prediction to reduce wire density, and verify buffer overflows. The main investigative work is to determine whether SMT solvers have been developed enough to be incorporated into the work flow of the current compiler project.

2. METHODS

In the next few subsections I will discuss the technologies and algorithms used in the process of the research project.

2.1 Background

There are techniques of finding the bit-widths of variables without using an SMT solver during the static analysis phase of compilation. These techniques involve some form of arithmetic operation on each of the values and forming a dependency mapping.

2.1.1 Interval Arithmetic

Interval arithmetic is a naive approach to predicting ranges of bit widths. The idea is: given some operation between two symbolic integers or numbers, perform a specific operation on the interval defined by the symbolic variable. An interval is defined as the range over two integer values, such as $x = [1, 2]$ where $x_{min} = 1$ and $x_{max} = 2$. The four operations defined in interval arithmetic? are:

- Addition -

  $$[x_{min}, x_{max}]+[y_{min}, y_{max}] = [x_{min}+y_{min}, x_{max}+y_{max}]$$

- Subtraction -

  $$[x_{min}, x_{max}]-[y_{min}, y_{max}] = [x_{min}-y_{max}, x_{max}-y_{min}]$$

- Multiplication -

  $$[x_{min}, x_{max}] - [y_{min}, y_{max}] =$$

  $$[min(x_{min}y_{min}, x_{max}y_{min}, x_{min}y_{max}, x_{max}y_{max}),$$

  $$max(x_{min}y_{min}, x_{max}y_{min}, x_{min}y_{max}, x_{max}y_{max})]$$

- Division -

  $$\frac{[x_{min}, x_{max}]}{[y_{min}, y_{max}]} = [x_{min}, x_{max}] * [\frac{1}{y_{min}}, \frac{1}{y_{max}}]$$

  where $0 \notin [y_{min}, y_{max}]$.

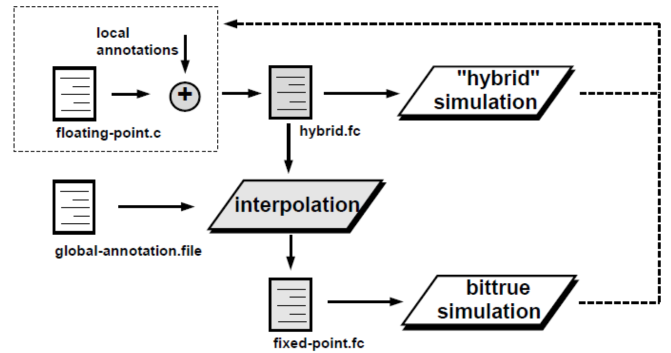

Figure 1: The interpolative step in the FRIDGE static analysis tool?

The FRIDGE system was a module added to the ANSI-C compiler to allow for an additional datatype known as "fixed" that specified,

- wl - The maximum value
- iwl - The minimum value
- sign - The integer sign of the value
- cast - How to handle overflow as well as quantization, such as rounding, truncation, or neither
- value - The C value that is being operated on

FRIDGE used an interpolative step during compilation to determine the ranges of the variables during compilation by first mapping all of the operations of the variables to a list and then analyzing the relationships between the variables over the course of compilation. This system is not robust enough to give reasonable approximations for the interval that variables exist because the operations it performs quickly increase in value beyond where they are useful. Multiplication often results in ranges that are grossly inflated over what the true range of the value is.

2.1.2 Affine Arithmetic

Affine arithmetic was another system used to try to interpolate values during compilation. It works by representing a variable as

a sum of constituents each with an independent uncertainty source that contributes to the total uncertainty of an output.

$$x\hat{=}x_0 + x_1\epsilon_1 + x_2\epsilon_2 + ... + x_n\epsilon_n$$

where $\epsilon_i \in [-1, 1]$.

The uncertainty can contribute to other symbols in the computation chain, keeping correlations between them. The same four operations as interval arithmetic are defined here.

- Addition/Subtraction -

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=0}^{n} (x_i \pm y_i)\epsilon i$$

- Constant Multiplication -

$$c\hat{x} = (cx_0) + \sum_{i=0}^{n} (cx_i)\epsilon i$$

- Constant addition/subtraction -

$$\hat{x} \pm c = (x_0 \pm c) + \sum_{i=0}^{n} (x_i)\epsilon i$$

- General Multiplication -

$$\hat{x}\hat{y} = (x_0 + \sum_{i=1}^{n} x_i\epsilon_i)(y_0 + \sum_{i=1}^{n} y_i\epsilon_i)$$

One such implementation of affine arithmetic is the MiniBit**?** compilation system. It performed two steps on the incoming program in order to determine the affine interval ranges.

- Precision Analysis - Analyzing the sensitivity of the output from a computation to slight changes in the bit widths.
- Range Analysis - Studying the data range of the computation and ensuring that the variables in the design have enough bits to accommodate this range.
- Then use the results of these computations in aggregate to find the optimal ranges of variables

Precision analysis allows us to determine which bit to round to when performing quantization. Effectively uses a fractional bit width to approximate how much of an exponent is required on top of the integer bit width already needed to represent the number. Affine arithmetic also suffers from explosion of values during the multiplication stage, and thus, while it offers a better approximation than interval arithmetic, does not offer an improvement that is as useful.

2.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) are an advancement made on SAT theorem provers. They allow a user to determine satisfiability with respect to some background theory that fixes the interpretations of certain predicate and function symbols. Applications of SMT often involve of more than one theory at a time. In these cases the satisfiability is understood as being modulo some combination of those various theories. Array, arithmetic, fixed-width bit-vectors, and inductive data types can all be represented using SMT**?**. What makes these theories useful is that it allows a compiler to generate a series of symbolic relationships between variables and verify that the relationships have a satisfiable nature to them.

2.3 Algorithms

2.3.1 Fibonacci

Fibonacci is a function of an integer input giving the result as a number following an integer sequence. It is define as:

$$fib(i) = fib(i-1) + fib(i-2)$$

Where $fib(0) = 0$ and $fib(1) = 1$. This creates the following sequence of integers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, ...$$

This function is very useful when testing verification algorithms because of its intensive recursive nature and its simplicity in representation. There is an iterative version of this algorithm which allows for a more stack friendly representation.

```
def fib(n):
        a, b = 0, 1
        for i in range(n):
                a, b = b, a + b
        return a
```

This python defined function iterates over all values from $0...n$ and proceeds to redefine a and b until the final value a is calculated.

2.3.2 Fir Filter

Finite impulse response (FIR) is a filter whose length of input response is of a finite duration**?**. This allows them to be a weighted sum of the most recent input values:

$$y[n] = b_0x[n] + b_1x[n-1] + ... + b_N[n-N] = \sum_{i=0}^{N} b_ix[n-i]$$

where:

- $x[n]$ is the input signal
- $y[n]$ is the output signal
- N is the filter order
- b_i is the value of the impulse response at the ith constant

This is a convenient representation of a summation function. A common problem when analyzing a FIR filter is to determine that if the input range is known, does the output range exceed some minimum and maximum value. For example, the input x may be known to be $-10 < x < 10 \forall x \in X$ where X is the input signal. Given the constant b_i, can the output be guaranteed to be within $-128 < y < 128 \forall y \in Y$ where Y is the output signal.

2.3.3 Bit Width Prediction

Bit width prediction is the static analysis of a program to determine the integer range that a value occupies in a program so as to reduce the space required to store that variable. The proposed algorithm takes a series of relationships and constraints on a series of variables determined via interval arithmetic and uses range refinement over the constraints to determine a well defined relationship between symbolic representations of variables. The process is divided into a two step process whereby a decision step selects a variable, splits the range of the variable into two, and then temporarily discards one of the sub-ranges. The second step, or propagation step, infers ranges of other variables from the newly split range**?**. The range refinement step comes after the propagation step occurs, whereby the range is analyzed using a binary search over the range of the variable. The SMT solver is then queried to determine if the new, smaller range for the variable is still satisfiable. Once the range is within a certain threshold, the result is confirmed and the routine exits.

In the paper they also highlight timeout as a major issue, as SMT solvers can typically run for long periods of time if given enough complex constraints. The researchers recommended a timeout of five minutes for their analysis, however it varies on the runtime system and the intensity of the constraints being evaluated. In the case of this paper, a timeout was not used in the code but an implicit timeout of 60 seconds was used when evaluating the data.

2.3.4 Buffer Overflow Checking

A given program written in an imperative language will often have array accesses or other buffer indexing that are vulnerable to invalid memory accesses by incorrect indexing or overflow. One

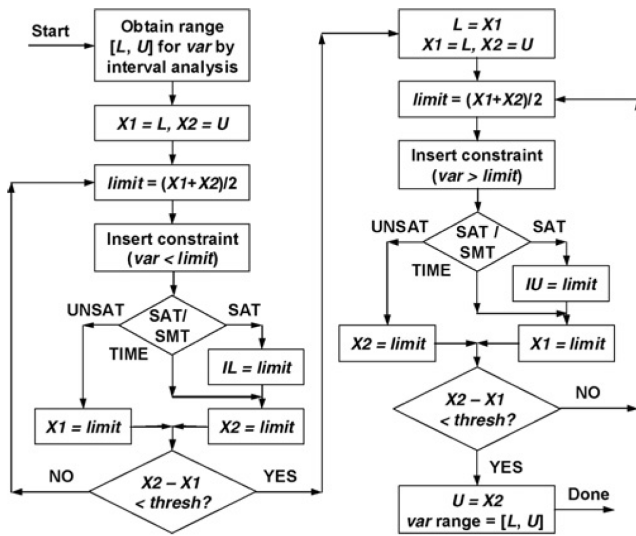

Figure 2: Bit width prediction using an SMT solver and an interval?

paper proposed a solution to this problem by building a system that would detect any potential buffer overflow events by performing a static analysis over a program, using either affine arithmetic or interval arithmetic, or get approximations of the ranges of an array or whether a pointer could be pointing to an invalid location in memory. They compiled these alarms into a list, and then using an SMT solver proceeded to refine the false alarms using a symbolic execution step that determined whether the input program was valid or not.

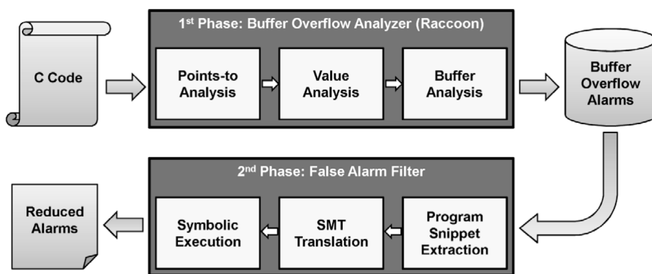

Figure 3: The Raccoon buffer overflow alarm detection algorithm with added SMT verification?

The first extracted the relevant program snippet that was causing the false alarm by using a backward slicing that only extended to the beginning of the current procedure. If the alarm relied on values outside of the context of the procedure, they gave up evaluating the alarm. The second step involved building an initial context formula from the given snippet. The variables obtained from the slice are given their initial values from either previous analysis or from the initialization of them in an earlier part of the code. The third step involves translating the relevant program snippet into SMT formulae. If a statement is encountered which cannot be translated to SMT formulae, it is mimicked using the interval results from the first stage analysis (interval arithmetic). Then the program is symbolically run through the SMT analyzer checking to see if the system is currently satisfiable or if the values can be adjusted using reasonable approximation to determine what might be causing the alarm. Finally, the alarm is filtered either by evaluating the expression as satisfiable, or by reporting the alarm as a true alarm?.

2.3.5 Loop Unrolling

Researchers at KAIST? gave, as part of their analysis on buffer overflow analysis, a method for loop unrolling. The method involved providing a given context based on the previous iteration of the loop and adding additional constraints based on the contents of the loop. This means that each iteration of the loop is evaluated to determine if it is a potential alarm that would prove unsatisfiable. This means translating each iteration of the loop into a series of SMT constraints and querying the SMT solver. On optimization over this method involves converting an entire loop into a series of constraints, providing a range of the inputs into the loop, and then determining if the intermediate values are violating any satisfiability constraints. Both methods are costly, but they each have their own benefits depending on whether memory space is being optimized or runtime is being optimized. In the case of runtime optimization, inputting a larger query once into an SMT solver will run much faster than inputting many smaller queries. In the case of memory optimization, the larger query will become quite expensive to create and hold in memory, but the smaller queries will be more manageable.

3. LOGISTICS

3.1 Timeline

The following timeline best captures the goals of each week and how they were completed. Meetings were held with Professor Stephen Edwards from January 21st until May 5th.

January 21st	Perform a literature review of integer prediction and floating point to fixed point conversion.
February 9th	Investigate which SMT solvers are most applicable to the research project.
February 18th	Investigate iterative fibonacci and array bounds checking with Z3.
February 25th	Investigate the Lonestar GPU benchmark as well as the AMD and NVidia pipeline.
March 1st	Look into how Microsoft's f* language works and find a way to query Z3 which specific requests. Also investigate how the FRIDGE system is implemented.
March 10th	Investigate arrays bounds checking using Z3.
April 7th	Implement bit width prediction algorithm using Z3.
April 14th	Retry the fibonacci unrolling using a slightly different way to check bounds.
April 21st	Look into Fir filters and bubble sort.
April 28th	Writeup and analysis.

4. Z3 TUTORIAL

Z3 is a state-of-the art theorem prover that is under constant development by Microsoft Research. It is used to check the satisfiability of a series of logical formulas and offers performant results over alternative SMT solvers. Because it is under constant development and has a prominent research backing, it makes it an attractive solution for SMT solving.

4.1 Prerequisites

The source for Z3 is stored in a Github repo here. Z3 is compatible with Windows and Linux and can be built with appropriate tools from either of those platforms. Follow the instructions on the corresponding Github repo to get the tool installed on your local machine.

4.2 Getting Started

Z3 uses a language known as smt-lib to define a series of functions to define a constraint over a system of equations. The entire language definition can be found at here.

4.2.1 Using Z3Py

Z3Py was the tool of choice for much of this research because it is convenient to use and also allows for rapid development of programs in Z3. It is a Python wrapper for Z3 that produces SMT-LIB syntax of series of logical theorems. The following is an example of a simple theorem finding the relationship between a, b, and c:

```
from z3 import *

a = Int('a')
b = Int('b')
c = Int('c')
s = Solver()
s.add(a == b + c)
s.add(b == 10 * c)
s.add(c == 12)

if s.check() == sat:
        print('verified')
else:
        print('unverified')
```

Z3Py has facilities for running the Z3 theorem solver directly within a Python program by calling the Z3 binaries and injecting the SMT-LIB syntax generated by the constraints inputted into the solver. The associated SMT-LIB syntax for the above code is:

```
(declare-fun c () Int)
(declare-fun b () Int)
(declare-fun a () Int)
(assert (= a (+ b c)))
(assert (= b (* 10 c)))
(assert (= c 12))
```

The syntax for SMT-LIB is reverse-polish notation. The guide for Z3Py can be found at `http://www.cs.tau.ac.il/~msagiv/courses/asv/z3py/guide-examples.htm`.

4.2.2 Basic Elements

Z3Py has several basic elements used for theorem proving.

Number types

- Real - A mathematical representation for all real numbers
- Int - A mathematical representation for all integers

These are both expressed by $Real$ and Int in Z3Py, respectively. Z3Py also has support for fixed point representation, allowing for analysis on the machine specific representation of floating point numbers. For the analysis used in this paper, an 8-bit mantissa and a 24-bit significand, but Z3 supports arbitrary precision.

The following are basic elements of the theorem prover

- ite - If Then/Else - Allows for the common programming paradigm of If Then/Else in Z3. Useful for emulating the common programming paradigm of If Then/Else.
- forall - For All - Over a specific range of an input symbol, the following constraint must be satisfied.

5. RESULTS AND ANALYSIS

5.1 Experiments

Several experiments were conducted to determine feasibility in Z3. These included implementing an iterative version of fibonacci, a range checker for a fir filter, and determining the range and bit widths of integer values in a series of expressions.

5.1.1 Fibonacci

There were many attempts to implement the recursive version of fibonacci using Z3. This initially began by trying to have Z3 determine the relationship between a python function and a generated output value. Unfortunately, because Z3Py relies on SMT-LIB to generate theories for satisfiability, using a recursive python function would not compile or generate the necessary code. Another attempt at trying to have Z3 interpret recursive functions involved using a ForAll constraints and defining the fibonacci function in terms of previous values. Z3 rejected every combination of constraints because the relationship between a variable and itself was not able to be resolved. Z3 does not currently support inductive proofs, which recursive fibonacci is, and therefore had to be converted into a format that was more friendly to Z3 syntax.

```
def fib_h(x, y, z):
        if z == 0:
                return x
        else:
                return fib_h(y, x + y, z-1)

def fib(x):
        return fib_h(0, 1, x)
...
for x in range(10):
        s.add(fib(x) == r)
```

Figure 4: Attempting to set fibonacci equal to some result intermediate value r

```
s.add(ForAll(i,Implies(And(i>=2, i<20),
                       fib(i)==fib(i-1)+fib(i-2))))
```

Figure 5: Using a ForAll constraint to define fibonacci as a recursive function.

The next strategy was to use an iterative version of fibonacci that only relied on previous intermediate. This would allow for a simple loop unrolling technique to be used, which meant carrying out the operations upon each iteration of the fibonacci function at each step. The first attempt of this was to generate a linear series of relationships between equations by defining each intermediate of fibonacci as a function of the previous set of intermediates. This involved using the iterative version of fibonacci defined in an earlier section of this paper. This worked and Z3 began producing the results of fibonacci at each step. The next step was to emulate control flow in Z3, and this was done using the ite directive. This allowed each iteration to be a function of whether the current iteration's index, i, is greater than some input parameter, n. By doing this, Z3 could determine whether to stop evaluating Z3 at a particular value or to continue with the next If statement.

```
If(ib < param,
   And(x0 == yb,
       y0 == zb,
       z0 == x0 + y0,
       i0 == ib + 1,
       If(i0 < param,
          And(x1 == y0,
              y1 == z0,
              z1 == x1 + y1,
              i1 == i0 + 1,
              And(param == i1, output == x1)),
          And(i0 <= param, output == x0))),
   And(ib <= param, output == xb))
```

Figure 6: Nested if statement controlled by a param variable

5.1.2 Fir Filter

The implementation for FIR filters was relatively straight forward. In order to evaluate whether a given output ever exceeded a maximum value or a minimum value, the output value was set equal to a series of additions. Because Z3 does not offer proper support for built in for loops, the summation had to be unrolled as either a

series of additions or a single addition. If this were a function being implemented, each iteration would need a flow control statement similar to the fibonacci example. The coefficients were constant values obtained through a sinc functions by taking $sin(x)/x$ from a range as a function of the minimum and maximum input parameters. This was also a test of fixed point evaluation in Z3, and it if was capable of giving reliable results despite needing to use high precision fixed point.

$$\begin{aligned} s_0 &= b_0 * x_n \\ s_1 &= b_1 * x_n + s_0 \\ s_2 &= b_2 * x_n + s_2 \\ &\dots \\ s_n &= b_n * x_n + s_{n-1} \\ output &= s_n \end{aligned} \tag{1}$$

Figure 7: Fir implemented as a series of additions

$$output = b_0 * x_n + b_1 * x_n + \dots + b_n * x_n \tag{2}$$

Figure 8: Fir implemented as a long, single addition

5.1.3 Bit Width Prediction

Using the bit-width calculation algorithm as mentioned in the Algorithms section of this paper, Z3 was tested for its accuracy and consistency of answers. A series of examples were taken from the paper that defined the algorithm and evaluated on the same input.

5.2 Analysis

The following timing and accuracy analysis were performed on an Ubuntu 15.10 VM running on an Intel i7-5820k with 32GB of RAM. Z3 was run on Python 2.7.10 using Z3 version 74.

5.2.1 Fibonacci Timing Analysis

For each of these tests, the constraints for Fibonacci were generated using some fixed maximum value (i+1) and checked to see if they ever exceeded the answer of $fib(i) + 1$ for each i. The results for Fibonacci were very predictable in the case where the loop was not unrolled. It quickly grew exponentially, and after $fib(200)$ takes far longer than a minute. Interestingly, the unrolled version of Fibonacci was not very predictable, likely due to the heuristic-based nature of Z3 optimizing the evaluation of the satisfiability equations.

What can be surmised from this is that unrolling the for loop can be an effective way of determining the output range of a function assuming the function has a reasonably small input range. The results shown above are repeatable for each input and very consistent.

5.2.2 Fir Filter Timing Analysis

For the FIR filter testing, the parameters used were that $-10 < x < 10$ and $-1000 < y < 1000$. For generating the parameters, a simple sinc function was used ranged over -500 to 500. The results found were that the behavior of a series of sums were more predictable and less erratic than the fir filter using a single sum of several variables. This is likely due to a heuristic that Z3 is using in the back end to solve the system of equations. Due to the intensive nature of fixed point numbers, the amount of time necessary to calculate the satisfiability of the system increased rapidly. The fixed point notation used was an 8-bit mantissa with a 24-bit significand.

5.2.3 Accuracy

The verification of Z3 using the results of a given paper were evaluated for consistency and validity. In this case three examples were used from *Bit-Width Allocation for Hardware Accelerators for Scientific Computing Using SAT-Modulo Theory*?,

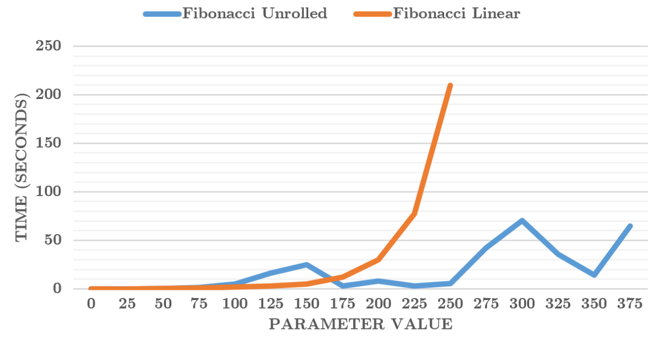

Figure 9: Fibonacci unrolled (blue) taking unpredictable amounts of time and iterative Fibonacci taking expected amounts of time (orange)

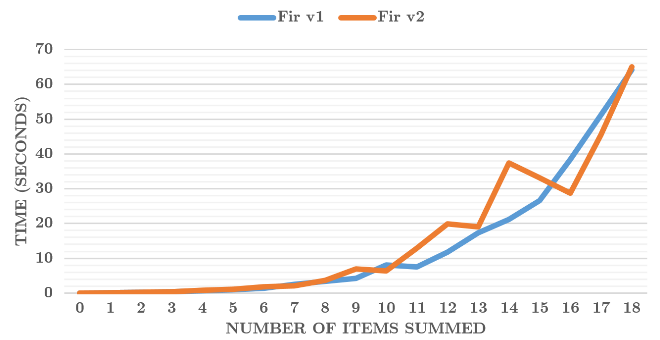

Figure 10: The implementation of fir filter using a series of sums (blue) is more predictable than the version of fir using a single sum (orange)

namely Rational Functions, Doppler effect, and a Relational example. The main difference between the paper and the analysis in this paper is that Z3 is being used which is a newer, more robust SMT solver. Other examples were omitted due to the lack of documentation in the paper for how they were implemented or that they were too similar to other examples. The paper presented a convenient way of representing division of two symbolic symbols, but assigning an intermediate value to arbitrate the division. For the sake of determining the difference in Z3 and what analysis they performed, the same technique was used. All values in the following examples are defined as real numbers.

For the relational example the following constraints were used:

```
x <= 20
x >= -20
y <= 20
y >= -20
q2 >= 0.01
q1 == x * y
q2 == x**2 + y**2
q1 == q2 * z
```

For the Rational example the following constraints were used:

```
q1 == 25 * t ** 2 + 125
q2 == t ** 2 + 1
z1 == q1/q2
q3 == -200 * t
q4 == q2 ** 2
```

output	HySAT SMT		Z3	
	range	bits	range	bits
z	$[-1, 1]$	2	$[-1, 1]$	2
q1	$[-400, 400]$	10	$[-400, 400]$	10
q2	$[0, 800]$	10	$[0, 800]$	10

Figure 11: Comparison of accuracy of Relational Function accuracy

```
z2 == q3 / q4
t <= 100
t >= -100
```

output	HySAT SMT		Z3	
	range	bits	range	bits
z1	$[24, 126]$	7	$[25, 125]$	7
z2	$[-67, 67]$	8	$[-65, 65]$	8
q1	$[124, 250126]$	18	$[125, 250125]$	18
q2	$[0, 10002]$	14	$[1, 10001]$	14
q3	$[-20001, 20001]$	16	$[-20000, 20000]$	16
q4	$[1, 100020001]$	27	$[1, 100020001]$	27

Figure 12: Comparison of accuracy of Rational Function accuracy

For the Doppler example the following constraints were used:

```
-30 <= T
T <= 50
20 <= v
v <= 20000
-100 <= u
u <= 100
q1 == 331.4 + 0.6 * T
q2 == q1 * v
q3 == q1 + u
q4 == q3 ** 2
z == q2/q4
```

output	HySAT SMT		Z3	
	range	bits	range	bits
z1	$[0, 138]$	8	$[0, 138]$	8
q1	$[313, 362]$	6	$[313, 362]$	6
q2	$[6267, 7228000]$	23	$[6268, 7228000]$	23
q3	$[213, 462]$	8	$[213, 462]$	8
q4	$[45539, 212890]$	18	$[45539, 212890]$	18

Figure 13: Comparison of accuracy of Doppler Function accuracy

In some cases, such as for q2 in the Doppler example, Z3 was able to give a slightly more precise range over the value. This is likely to the precision value they were using during testing, which was probably set to 2 whereas during my testing it was set to 1, thus giving a slightly more accurate result.

5.3 Next Steps

Next steps for this analysis include providing a more robust analysis of sorting algorithms and determining how to properly analyze them using Z3. This would likely include analyzing whether there are any buffer overflows during sorting. It would also involve verifying that an array is sorted as a result of sorting. Also critical is generating a general purpose algorithm for converting loops and statements written in an arbitrary language (such as Haskell) and converting them into Z3 constraints and how to perform the static analysis. A method of loop unrolling is proposed in this paper, but that is not necessarily generalizable. Other efforts would include converting more functions into Z3 syntax to see how they translate into an SMT syntax. For the fixed point analysis, research should be done to determine the impact that the fixed point has on the amount of time Z3 spends to determine the satisfiability of a system.

6. CONCLUSION

Through the use of an SMT solver (Z3), a program can be accurately analyzed using static analysis. This includes determining variable bit widths and checking for buffer overflow. The current best techniques to do this involve loop unrolling and initially performing interval arithmetic to get an approximation of the range of the variables. Future analysis in this topic would include determining how much of a program can Z3 reliably generate on its own and to what extent can Z3 accurately determine a system. For the purposes of a basic compiler, however, Z3 has a toolkit that is applicable to solve most relational problems.

7. REFERENCES

Sanjit A. Seshia Clark Barrett, Roberto Sebastiani and Cesare Tinelli. *Handbook of Satisfiability.* IOS Press, 2008.

H. Keding, M. Willems, M. Coors, and H. Meyr. Fridge: a fixed-point design and simulation environment. In *Design, Automation and Test in Europe, 1998., Proceedings*, pages 429–435, Feb 1998.

Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. Filtering false alarms of buffer overflow analysis using {SMT} solvers. *Information and Software Technology*, 52(2):210 – 219, 2010.

A. B. Kinsman and N. Nicolici. Bit-width allocation for hardware accelerators for scientific computing using sat-modulo theory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):405–413, March 2010.

D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1990–2000, Oct 2006.

FIR Filter Properties. http://dspguru.com/dsp/faqs/fir/properties.

Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. Hardware synthesis from a recursive functional language. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '15, pages 83–93, Piscataway, NJ, USA, 2015. IEEE Press.

8. CODE LISTING

8.1 Fibonacci Sum

```python
from z3 import *
import time
import sys


def unrollfib_h(param, output, cur_level, max_level, x_old, y_old, z_old, i_old):
    i = Int('i' + str(cur_level))
    z = Int('z' + str(cur_level))
    y = Int('y' + str(cur_level))
    x = Int('x' + str(cur_level))

    update = And(x == y_old, y == z_old, z == x+y, i == i_old + 1)
    ifcond = And(i_old < param, update)
    varcond = And(i_old == param, update)

    if cur_level == max_level:
        return And(varcond, output == x_old)

    return If(ifcond, unrollfib_h(param, output, cur_level + 1, max_level, x, y, z, i), And(varcond, output == x_old))

def unrollfib(maxval):
    param = Int('param')
    output = Int('output')
    fib = {}

    x = Int('xb')
    y = Int('yb')
    z = Int('zb')
    i = Int('ib')

    fib["init"] = [x == 0, y == 1, z == 1, i == 0]
    fib["body"] = unrollfib_h(param, output, 0, maxval, x, y, z, i)
    fib["param"] = param
    fib["output"] = output

    return fib

def unrollfibto(start=0, finish=30, maxval=1000, unrollto=200, verbose=True):
    fib = unrollfib(unrollto)
    s = Solver()
    s.add(fib["init"])

    for i in range(start, finish+1):
        s.push()
        s.add(fib["param"] == i)
        s.add(fib["output"] < maxval)
        s.add(fib["body"])
        s.check()

        # if s.check() == unsat:
        #     print("Solution is not valid")
        #     print("fib(" + str(i) + ") is not less than " + str(maxval))
        #     return
        # else:
        #     print("fib(" + str(i) + ") == " + str(s.model()[fib["output"]]))

        s.pop()

    # print("Fibonacci from ", str(start), " to ", str(finish), " is less than ", str(maxval))
```

```
60
61 def main():
62     a = 0
63     b = 1
64
65     for i in range(0, 200000, 25):
66         t0 = time.time()
67         unrollfibto(finish=i, maxval=a + 1,
   ↪   unrollto=i+1, verbose=False)
68         t1 = time.time()
69
70         a, b = b, a + b
71
72         total = t1-t0
73         print(str(i) + " " + str(total))
74
75
76 main()
```

8.2 Fibonacci Unrolled For

```
1  from z3 import *
2  import time
3  import sys
4
5  def unrollfib_h(param, output, max_level):
6      i = Int('i' + str(max_level))
7      z = Int('z' + str(max_level))
8      y = Int('y' + str(max_level))
9      x = Int('x' + str(max_level))
10
11     i_old = Int('i' + str(max_level - 1))
12     x_old = Int('x' + str(max_level - 1))
13     y_old = Int('y' + str(max_level - 1))
14     z_old = Int('z' + str(max_level - 1))
15
16     prev_case = And(param == i_old, output ==
   ↪   x_old)
17
18     for k in range(max_level - 2, -1, -1):
19         i = i_old
20         z = z_old
21         y = y_old
22         x = x_old
23
24         i_old = Int('i' + str(k))
25         x_old = Int('x' + str(k))
26         y_old = Int('y' + str(k))
27         z_old = Int('z' + str(k))
28
29         ifcond = i_old < param
30         elsecase = And(i_old <= param, output ==
   ↪   x_old)
31         thencase = And(x == y_old, y == z_old, z
   ↪   == x+y, i == i_old + 1, prev_case)
32         prev_case = If(ifcond, thencase,
   ↪   elsecase)
33
34     i = i_old
35     z = z_old
36     y = y_old
37     x = x_old
38
39     i_old = Int('ib')
40     x_old = Int('xb')
41     y_old = Int('yb')
42     z_old = Int('zb')
43
44     ifcond = i_old < param
45     elsecase = And(i_old <= param, output ==
   ↪   x_old)
46     thencase = And(x == y_old, y == z_old, z ==
   ↪   x+y, i == i_old + 1, prev_case)
47     prev_case = If(ifcond, thencase, elsecase)
```

```
48
49      return prev_case
50
51
52  def unrollfib(maxval):
53      param = Int('param')
54      output = Int('output')
55      fib = {}
56
57      x = Int('xb')
58      y = Int('yb')
59      z = Int('zb')
60      i = Int('ib')
61
62      fib["init"] = [x == 0, y == 1, z == 1, i ==
    ↪   0]
63      fib["body"] = unrollfib_h(param, output,
    ↪   maxval)
64      fib["param"] = param
65      fib["output"] = output
66
67      return fib
68
69  def unrollfibto(start=0, finish=30, maxval=1000,
    ↪   unrollto=200, minval=0, verbose=True):
70      fib = unrollfib(unrollto)
71      s = Solver()
72      s.add(fib["init"])
73
74      s.push()
75      s.add(fib["param"] >= start)
76      s.add(fib["param"] <= finish)
77      s.add(fib["body"])
78      s.add(Or(fib["output"] > maxval,
    ↪   fib["output"] < minval))
79      s.check()
80
81      if(verbose):
82          if s.check() == unsat:
83              print("Solution is valid")
84              print("fib(" + str(finish) + ") is
    ↪   less than " + str(maxval))
85              # print(fib["body"])
86              return
87          else:
88              print("Solution is not valid")
89              print("fib(" +
    ↪   str(s.model()[fib["param"]]) + ") == " +
    ↪   str(s.model()[fib["output"]]))
90              print("This is either greater than "
    ↪   + str(maxval) + " or less than " +
    ↪   str(minval))
91
92      s.pop()
93
94  def main():
95      a = 0
96      b = 1
97
98      for i in range(0, 5, 1):
99          t0 = time.time()
100         unrollfibto(finish=i, maxval=a + 1,
    ↪   minval=0, unrollto=i+1, verbose=False)
101         t1 = time.time()
102
103         a, b = b, a + b
104
105         total = t1-t0
106         print(str(i) + " " + str(total))
107
108
109 main()
```

8.3 Fir Unrolled v1

```
1  import math
2  from z3 import *
3  import time
4  import sys
5
6  def gen_fir(coefficients, num, param):
7      constraints = []
8      var_list = []
9      stmt = 0
10
11     for i in range(num):
12         var = FP('s' + str(i), FPSort(8, 24))
13         var_list.append(var)
14         constraints.append(var == coefficients[i]
   ↪   * param)
15         stmt += var
16
17     output = FP('output', FPSort(8, 24))
18     constraints.append(output == stmt)
19
20     return output, constraints
21
22
23 def gen_coefficients(num=1000):
24     a = []
25     for x in range(0, num):
26         op = float(x - num/2)/10.0
27         if op == 0.0:
28             op = 1.0
29         val = math.sin(op)/op
30         a.append(val)
31
32     return a
33
34
35 def unroll_fir(x_max=100):
36     param = FP('param', FPSort(8, 24))
37
38     a = gen_coefficients(num=x_max)
39     output, constraints = gen_fir(a, x_max,
   ↪   param)
40
41     fir = {}
42
43     fir["body"] = constraints
44     fir["param"] = param
45     fir["output"] = output
46
47     return fir
48
49
50 def main(param_low, param_high, out_low,
   ↪   out_high, depth, inc):
51     for i in range(0, depth, inc):
52         t0 = time.time()
53
54         fir = unroll_fir(i)
55         s = Solver()
56         s.add(fir["body"])
57         s.add(And(fir["param"] >= param_low,
   ↪   fir["param"] <= param_high))
58         s.add(Or(fir["output"] > out_high,
   ↪   fir["output"] < out_low))
59         s.check()
60         # print(s.sexpr())
61
62         # if s.check() == unsat:
63         #     print("Fir filter is between -128
   ↪   and 128")
64         # else:
65         #     print("Fir filter failed")
```

```
        #     float_val =
↪   (s.model()[fir["output"]])
        #     float_val =
↪   float(float_val.significand()) * (2**
↪   float(float_val.exponent()))
        #     print("Result: ", str(float_val))

        t1 = time.time()
        print(str(i) + " " + str(t1-t0))


if len(sys.argv) != 7:
    print("Not enough arguments...")
    exit(0)

plow = float(sys.argv[1])
phigh = float(sys.argv[2])
olow = float(sys.argv[3])
ohigh = float(sys.argv[4])
num = int(sys.argv[5])
inc = int(sys.argv[6])

main(plow, phigh, olow, ohigh, num, inc)
```

8.4 Fir Unrolled v2

```
import math
from z3 import *
import time
import sys

def gen_fir(coefficients, num, param):
    constraints = []
    var_list = []
    stmt = 0

    for i in range(num):
        var = FP('s' + str(i), FPSort(8, 24))
        var_list.append(var)
        # constraints.append(var ==
↪   coefficients[i] * param)
        stmt += coefficients[i] * param

    output = FP('output', FPSort(8, 24))
    constraints.append(output == stmt)

    return output, constraints


def gen_coefficients(num=1000):
    a = []
    for x in range(0, num):
        op = float(x - num/2)/10.0
        if op == 0.0:
            op = 1.0
        val = math.sin(op)/op
        a.append(val)

    return a


def unroll_fir(x_max=100):
    param = FP('param', FPSort(8, 24))

    a = gen_coefficients(num=x_max)
    output, constraints = gen_fir(a, x_max,
↪   param)

    fir = {}

    fir["body"] = constraints
    fir["param"] = param
    fir["output"] = output

```

```
47      return fir
48
49
50  def main(param_low, param_high, out_low,
    ↪   out_high, depth, inc):
51      for i in range(0, depth, inc):
52          t0 = time.time()
53
54          fir = unroll_fir(i)
55          s = Solver()
56          s.add(fir["body"])
57          s.add(And(fir["param"] >= param_low,
    ↪   fir["param"] <= param_high))
58          s.add(Or(fir["output"] > out_high,
    ↪   fir["output"] < out_low))
59          s.check()
60          # print(s.sexpr())
61
62          # if s.check() == unsat:
63          #     print("Fir filter is between -128
    ↪   and 128")
64          # else:
65          #     print("Fir filter failed")
66          #     float_val =
    ↪   (s.model()[fir["output"]])
67          #     float_val =
    ↪   float(float_val.significand()) * (2**
    ↪   float(float_val.exponent()))
68          #     print("Result: ", str(float_val))
69
70          t1 = time.time()
71          print(str(i) + " " + str(t1-t0))
72
73
74  if len(sys.argv) != 7:
75      print("Not enough arguments...")
76      exit(0)
77
78  plow = float(sys.argv[1])
79  phigh = float(sys.argv[2])
80  olow = float(sys.argv[3])
81  ohigh = float(sys.argv[4])
82  num = int(sys.argv[5])
83  inc = int(sys.argv[6])
84
85  main(plow, phigh, olow, ohigh, num, inc)
```

8.5 Simple Bubble Sort

```
1   from z3 import *
2   import time
3   import sys
4
5   s = Solver()
6
7   length = Int('length')
8   index = Int('index')
9   s.add(ForAll(index, Implies(And(index <= 100,
    ↪   index >= 20), index <= length)))
10  s.add(And(index <= 100, index >= 0))
11  s.add(And(length <= 100, length >= 0))
12
13  print (s.check())
14  print(s.model())
```

8.6 Bit Width Prediction

```
1   from z3 import *
2   from math import log, ceil
3   import sys
4
5   def refine(solver, upper, lower, function):
6       THRESHOLD = 1
```

```
7
8       U = upper
9       L = lower
10      x2 = U
11      x1 = L
12      solver.check()
13
14      #Check upper limit
15      while x2 - x1 > THRESHOLD:
16          limit = (x2 + x1)/2
17          solver.push()
18
19          solver.add(function < limit)
20
21          if solver.check() == sat:
22              x2 = limit
23          else:
24              x1 = limit
25          solver.pop()
26          # print limit
27
28      # x1 = 10
29      L = x1
30      x2 = U
31      x1 = L
32      #Check lower bound
33      while x2 - x1 > THRESHOLD:
34          limit = (x2 + x1)/2
35          solver.push()
36
37          solver.add(function > limit)
38
39          if solver.check() == sat:
40              x1 = limit
41          else:
42              x2 = limit
43          solver.pop()
44          # print limit
45
46      U = x2
47      return U, L
48
49  class Model:
50      def __init__(self, name):
51          self.args = []
52          self.internals = []
53          self.constraints = []
54          self.outputs = []
55          self.name = name
56
57      def addConstraint(self, z3expr):
58          self.constraints.append(z3expr)
59
60      def addArg(self, arg):
61          self.args.append(arg)
62
63      def addOutput(self, output):
64          self.outputs.append(output)
65
66      def addInternal(self, internal):
67          self.internals.append(internal)
68
69      def getSolver(self):
70          s = Solver()
71          for constraint in self.constraints:
72              s.add(constraint)
73          return s
74
75      def range(self):
76          s = self.getSolver()
77          if s.check() == sat:
78              print "outputs for: " + self.name
79              for output in self.outputs:
```

```
80                  U, L = refine(s, sys.maxint,
    ↪  -sys.maxint, output)
81                  width = U - L
82                  print str(output) + " Range: (" +
    ↪  str(L) + ", " + str(U) + ") Bits: " +
    ↪  str(ceil(log(width, 2)))
83              print "Internals for: " + self.name
84              for internal in self.internals:
85                  U, L = refine(s, sys.maxint,
    ↪  -sys.maxint, internal)
86                  width = U - L
87                  print str(internal) + " Range: ("
    ↪  + str(L) + ", " + str(U) + ") Bits: " +
    ↪  str(ceil(log(width, 2)))
88          else:
89              print self.name + " is not
    ↪  satisfiable"
90
91  def example1():
92      ex1 = Model("example1")
93
94      z = Real("z")
95      x = Real("x")
96      y = Real("y")
97      q1 = Real("q1")
98      q2 = Real("q2")
99
100     ex1.addArg(x)
101     ex1.addArg(y)
102     ex1.addInternal(q1)
103     ex1.addInternal(q2)
104     ex1.addOutput(z)
105
106     # #c1
107     ex1.addConstraint(x <= 20)
108     ex1.addConstraint(x >= -20)
109     #c2
110     ex1.addConstraint(y <= 20)
111     ex1.addConstraint(y >= -20)
112     # c3
113     ex1.addConstraint(q2 >= 0.01)
114     #c4
115     ex1.addConstraint(q1 == x * y)
116     #c5
117     ex1.addConstraint(q2 == x**2 + y**2)
118     ex1.addConstraint(q1 == q2 * z)
119     #c6
120     # ex1.addConstraint(ForAll([x, y], 0 ==
    ↪  f(x,y)))
121
122     return ex1
123
124 def dopplerexample():
125     z = Real("z")
126     q1 = Real("q1")
127     q2 = Real("q2")
128     q3 = Real("q3")
129     q4 = Real("q4")
130     T = Real("T")
131     v = Real("v")
132     u = Real("u")
133
134     ex1 = Model("dopplerexample")
135
136     ex1.addArg(T)
137     ex1.addArg(v)
138     ex1.addArg(u)
139     ex1.addOutput(z)
140     ex1.addInternal(q1)
141     ex1.addInternal(q2)
142     ex1.addInternal(q3)
143     ex1.addInternal(q4)
144
```

```
    ex1.addConstraint(q1 == 331.4 + 0.6 * T)
    ex1.addConstraint(q2 == q1 * v)
    ex1.addConstraint(q3 == q1 + u)
    ex1.addConstraint(q4 == q3 ** 2)
    ex1.addConstraint(z == q2/q4)

    #Parameters
    ex1.addConstraint(-30 <= T)
    ex1.addConstraint(T <= 50)
    ex1.addConstraint(20 <= v)
    ex1.addConstraint(v <= 20000)
    ex1.addConstraint(-100 <= u)
    ex1.addConstraint(u <= 100)

    return ex1

def rationalexample():
    ex1 = Model('rationalexample')

    z1 = Real("z1")
    z2 = Real("z2")
    q1 = Real("q1")
    q2 = Real("q2")
    q3 = Real("q3")
    q4 = Real("q4")
    t = Real("t")

    ex1.addArg(t)
    ex1.addOutput(z1)
    ex1.addOutput(z2)
    ex1.addInternal(q1)
    ex1.addInternal(q2)
    ex1.addInternal(q3)
    ex1.addInternal(q4)

    ex1.addConstraint(q1 == 25 * t ** 2 + 125)
    ex1.addConstraint(q2 == t ** 2 + 1)
    ex1.addConstraint(z1 == q1/q2)
    ex1.addConstraint(q3 == -200 * t)
    ex1.addConstraint(q4 == q2 ** 2)
    ex1.addConstraint(z2 == q3 / q4)
    ex1.addConstraint(t <= 100)
    ex1.addConstraint(t >= -100)

    return ex1

def handle_function(function_model):
    s = function_model.getSolver()
    function_model.range()

def main():
    handle_function(example1())
    handle_function(rationalexample())
    handle_function(dopplerexample())

main()
```