Fuzzing Program Models with Satisfiability Modulo Theories (SMT) Solvers

WILSON NGUYEN, Stanford University

**Abstract:** A framework was developed to model programs for the sole purpose of finding inputs that would cause program crashes. The framework was developed in the SMT-LIB v2 input and output language for SMT solvers. Programs were limited in scope to those solely contain integers, booleans, integer operations, random access arrays, array index retrievals, if and else statements, and for-loop constructs. Once programs were modeled, assertions would be created to constrain the scope of possible inputs and assert that program models would evaluate to a ‘Crash’ type. The Z3 theorem prover by Microsoft Research was used to check the satisfiability of these assertions and return a model satisfying all the assertions. The model returned would contain inputs that would crash the modeled-program. The framework is currently successful at finding inputs that cause crashes by undefined integer operations and array index out of bounds retrievals.

CS Concepts: • **Programming Languages → SMT Solving**; *Z3 theorem prover*; • **Program Modeling → Fuzzing**; crashes

KEYWORDS

Fuzzing, SMT, Satisfiability Modulo Theories, Z3, Crashes

1 INTRODUCTION remember to cite Wikipedia for smt, fuzzing, constraint, smtlibv2

Fuzzing is a software testing technique meant to reveal inputs that cause the software being tested to crash, leak memory, or unintentionally modify protected program memory. Currently, popular fuzzing engines such as SAGE and Triton use a mixture of randomized inputs and constraint solving to reveal crash-causing inputs. This framework solely uses constraint solving to find these crash-causing inputs. Fuzzing assists software developers to find security bugs in their code and to address the memory and logical safety of certain programming languages.

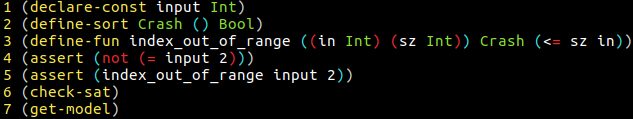
Satisfiability Modulo Theories or SMT problems are constraint satisfaction problems modeled in logical formulas incorporating theories expressed in first order logic. Theories used in this framework include Ints, ArraysEx, and Core theory (basic Boolean operators and functions). SMT solvers take in a set of assertions (logical formulas/constraints) and attempt to find a model that satisfies all of these assertions. This relates to the constraint-programming paradigm in which relations between constants and variables are expressed as constraints.

2 PROCESS AND EXPERIMENTATION DETAILS

2.1 Learning SMT-LIB v2 and Z3, Basic fuzzing

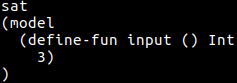
Before this project, I had no experience with SMT, Z3, or fuzzing. Thus, I had to learn both the strategies related to constraint solving and the SMT-LIB v2 language used to interact with SMT solvers (specifically Z3). After the initial portion of the rise4fun Z3 tutorial, I was able to develop this very crude, simple fuzzer for finding an index out of range (code on following page).

My initial approach was to define a datatype called *Crash,* which was a boolean. A *Crash* could be true or false. The constant *input* was an unassigned integer. The function *index\_out\_of\_range* took in two integers: *in,* an index, *sz*, the size of the array. *index\_out\_of\_range* would return a *Crash* type which would be true if the size of the array was less than or equal to the index and false otherwise. I created two assertions: the first was a formula that would be true if *input* was not equal to 2; the second was the function *index\_out\_of\_range* called on the arguments *input* and 2.



*A first crude fuzzing attempt.*

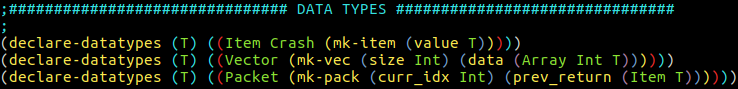
Checking satisfiability guarantees that there exists a model where all assertions could be satisfied (the logical formulas in the assertions all evaluate to true). The command *get-model* returns a model that satisfies all the assertions; in this case, the model defines *input* as a function that takes in no arguments and returns an integer value of 3. Everything in SMT-LIB v2 is a function. Thus, constants are just syntactic sugar for functions. In context, Z3 found an index, which was 3, that would cause an out of bounds array access, on an array of size 2.



*Z3 output in SMT-LIB v2*

2.2 Defining Datatypes

After completing the rise4fun Z3 tutorial and reading the documentation for SMT-LIB, I began to write the framework for what would ultimately become the core of my SMT fuzzer. The first step was to create datatypes that would be the foundational types for input and outputs in the program modeling.



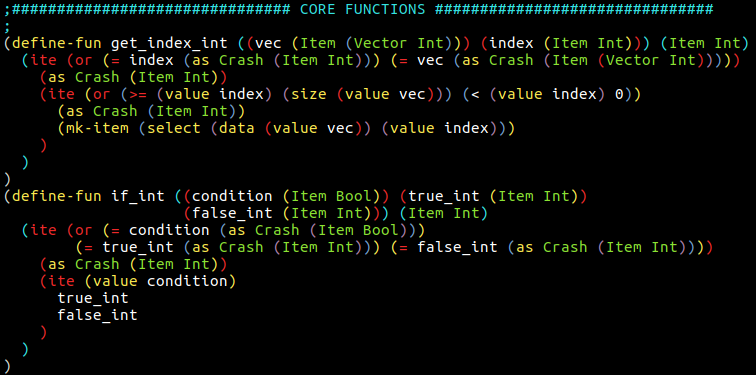
*Declaration of Item, Vector, and Packet datatypes.*

Inspired by exceptions and some-types, I envisioned that all functions would receive as input either a crash or a valid value. If a function received a crash, then it would in turn return a crash that would cause surrounding functions to return crashes (carrying crashes up the chain). Thus, I defined an *Item* type that was either a *Crash* or contained a *value* of type *T*. The symbol *T* is a placeholder for a type; therefore, *Items* can contain values of any type. Since everything in SMT-LIB is a function, *Arrays* are simply functions that take in an integer (the index) and return a value (the value at the index). Since I was testing out of bounds access, I needed a datatype that contained both *Array* functionality and a limited size. Thus, I defined the type *Vector,* which contained both an integer *size*, and *data* an *Array* *Int T*. Thus, data would extract a function that takes in an integer and returns a value of type *T*. Finally, in order to model for-loops, I needed a datatype that would package up the current index and the value returned by the previous iteration. Thus, the datatype *Packet* contains an integer *curr\_idx* and an *Item* *T* retrieved by *prev\_return*.

*Note: In these datatype declarations, value, size, data, curr\_idx, prev\_return are selector functions that extract values out of the datatypes and mk-item, mk-vec, mk-pack are constructors.*

2.3 Core Functions

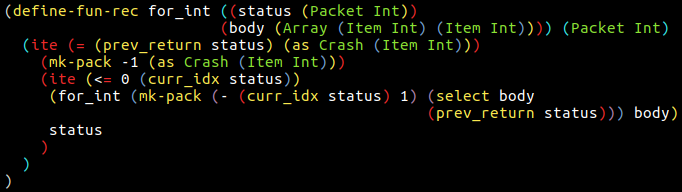
I decided that essential expressions to include in the framework were index access, if/else statements, and for-loops. Thus, I defined functions that would emulate these and propagating crashes when necessary.



*Model for array index retrieval and if/else statements*

The function *get\_index\_int* takes in as arguments an *Item* *(Vector Int)* assigned to *vec* and an *Item Int* assigned to *index* and returns an *Item Int*. The command *ite* *(boolean expression) (expression a) (expression b)* returns *expression a* if the *boolean expression* is true and *expression b* otherwise. Thus, *get\_index\_int* returns a *Crash* when vec or index are *Crash*, the value of *index* is greater than or equal to the size of *vec*, and when the value of *index* is less than zero. If *get\_index\_int* does not return a *Crash*, it returns an *Item Int*, which is the value at the arg *index*.

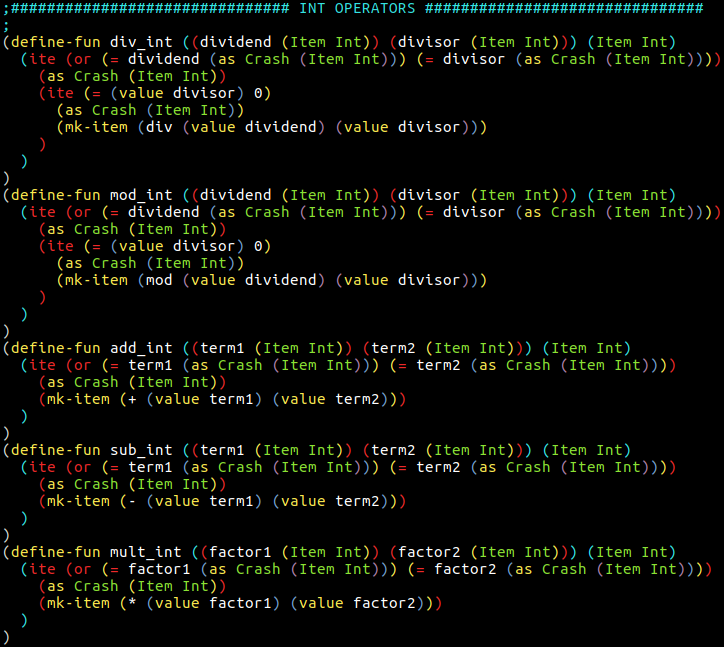
The function *if\_int* takes in as arguments an *Item Bool* as condition, *Item Int* as *true\_int*, and *Item Int* as *false\_int.* If any of the input parameters are *Crash*, then the if/else statement will return a crash. Otherwise, *if\_int* will return the appropriate *Item Int:* *true\_int or false\_int.*



There were three implementations of the for-loop function each with their own tradeoffs (recursive function theories are on the very limit of what Z3 can handle). The one presented in this section was the most generic (other versions are described in the results discussion section). Conceptually, I modeled a for-loop as a recursive function that takes in as arguments a status (containing the current index and the return value of the previous iteration) and a function that serves as the body of the for-loop. Since SMT-LIB does not have theories that support higher-order functions, I decided to use the theories of Arrays to emulate a lambda function as an argument (since arrays are just syntactic sugar for a function). The *for\_int* propagates a *Crash* when the previous iteration returned a *Crash*. Otherwise, if the current index is greater than or equal to zero, recurse further with a new status that contains 1 minus the current index and the value returned by the body function evaluated on the value of the previous iteration. Otherwise, if the current index is less than zero, return the last status.

2.4 Integer Operations

Up until this stage, the only crashes modeled are those caused by index out of bounds during array retrievals. To further the model, I wanted to build on the theory of Integers. *Crash* needs to be propagated up the chain during integer operations. Additionally, division or mod with a dividend of zero should result in a crash.

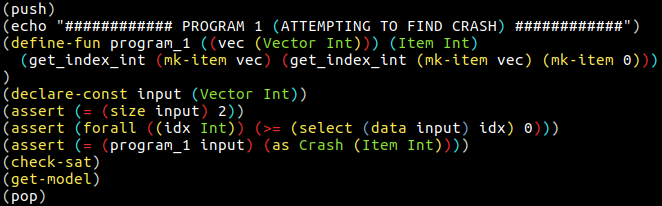


*Integer operator definitions*

Here I defined the integer operations as expected. All of these functions return a *Crash* if any of the input *Item Int* s are of the *Crash* type. Additionally for the case of mod and division, if the dividend item has the value of 0, then it should return a crash. Otherwise, the functions return what is expected from their operation.

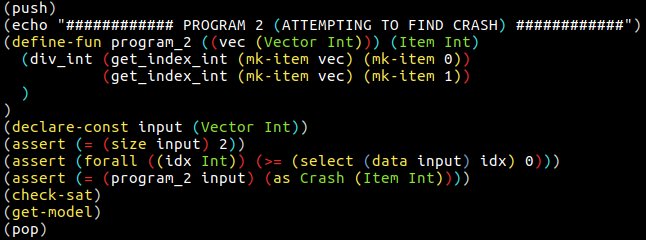
2.5 Program Models

As suggested by Varun (CS242 TA), I decided to limit the scope of programs to those only composed of the datatypes, core functions, and operations defined above. Additionally, programs are set to take in an arbitrary length vector (unless restricted by an assertion) and eventually return a single integer. Thus, programs are defined as functions that take in a *Vector Int* and return an *Item Int,* which could be a *Crash* type. Below is the first program definition.



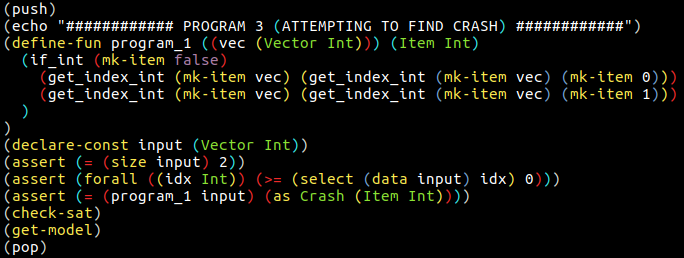
*Program 1.*

Z3 maintains a global stack of all user declared formulas and assertions. In order to prevent definition and declaration conflicts, I decided to model each program within its own stack scope with the push and pop SMT-LIB commands. The function *program\_1* models a simple program that returns the integer found at the index, which is the integer at index 0. In a more familiar syntax, *program\_1* attempts to model *vec[vec[0]]*. The input to the program is a constant declared as *input* of type *Vector Int*. Constants are functions that are not predefined (thus, Z3 must find a model containing a definition for this *input*). I added a couple of additional assertions to limit the scope of the input, just to address cases where the limited scope is necessary. In this case, the *input* must have a size of 2 and all elements in the *Vector* must be greater than or equal to zero. The final assumption is that the return of *program\_1* called on *input* is equal to *Crash*. The commands *check-sat* and *get-model* check to see if all assertions can be satisfied and retrieve a model that does so.

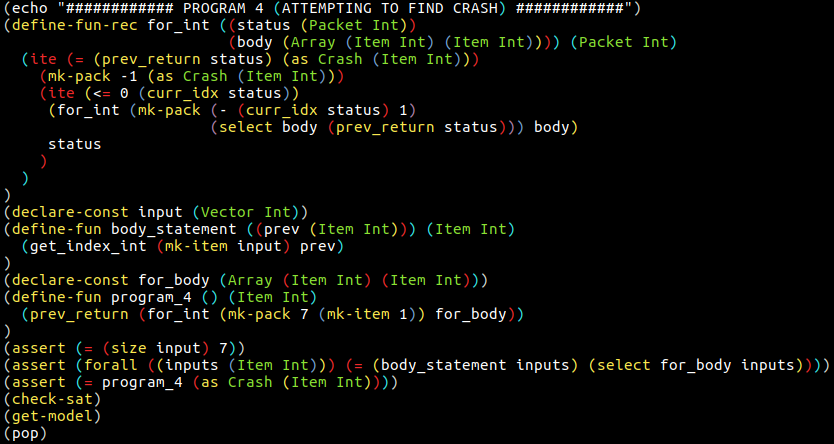


*Program 2.*

To demonstrate the ability of the fuzzer to find crashes related to integer operations, I model the program: *vec[0]/vec[1]* with similar assertions on the input.



To test the functionality of if statements, I modeled the program: *if false: vec[vec[0]] else: vec[vec[1]].*



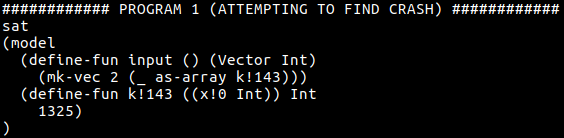
*Program 4.*

Finally, to test the functionality of for-loops, I created *program\_4*. This is one of the 3 different implementations of for-loops (later versions discussed in results and discussion). First, I declare the constant *input* as a *Vector Int* with a later assertion that it must have a size of 7. The function *body\_statement* contains the statements that are contained in the for-loop body. In this case, the body models this: *vec[prev]* where prev is the *Item Int* returned by the previous iteration. Since Z3 does not support higher-order functions (it does not have those theories) and has trouble with macros, I had to declare a constant *for\_body* that is an *Array (Item Int) (Item Int)* and assert that for all possible indexes, the return of *for\_body* would be equal to the return of *body\_statement.* This was meant to imitate the functionality of lambda functions in SMT-LIB. Finally, *program\_4* is defined as the last return of the recursive call to *for\_int* that starts off with a status of current index 7 and a previous return value of 1. The body of the for-loop is *for\_body*. The final assertion is that *program\_4* returns a *Crash*.

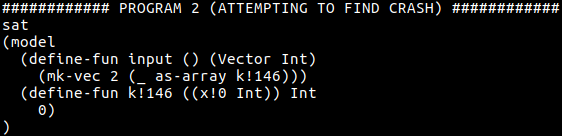
3 RESULTS AND DISCUSSION

3.1 Results of SMT Solving (Z3) - Success

Every collection of assertions, declarations, and definitions (in the push and pop stack scopes) were both satisfiable and produced valid models that would have resulted in crashes. The output of Z3 below states that a vector of size 2 that contains an array that always returns 1325 will crash the program. Thus, *vec[vec[0]]* will crash if the input vector given is *[1325, 1325].*

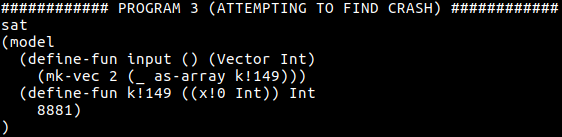


The output of Z3 states that an input vector of *[0,0]* will crash the program *vec[0]/vec[1].*



*Z3 output on Program\_2*

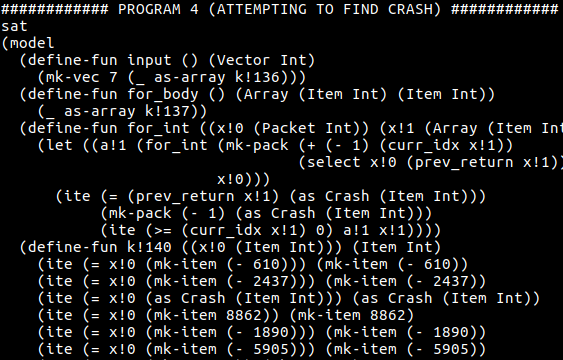
The output of Z3 for program 3 states that an input vector of *[8881, 8881]* will crash the program *if false: vec[vec[0]] else: vec[vec[1]].* It is important to note that Z3 did not have to produce a model containing both elements in the vector having the same value. As can be seen in the for-loop program (discussed below), Z3 returns the first model that it finds that satisfies all the assertions.



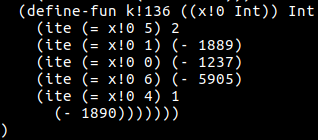
*Z3 output on Program\_3*

3.2 For-loops - Z3 / SMT Limits

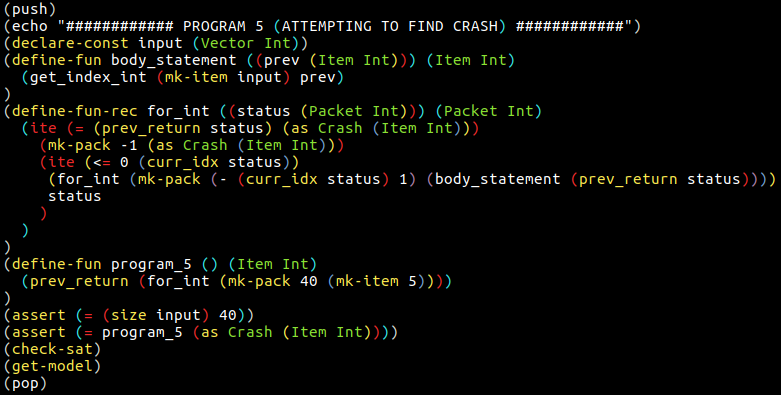
Since there were 2 constants in *program\_4*, *for\_body* and *input*, which were *Array (Item Int) (Item Int)* and V*ector,* respectively. Z3 produced a model containing definitions for both.



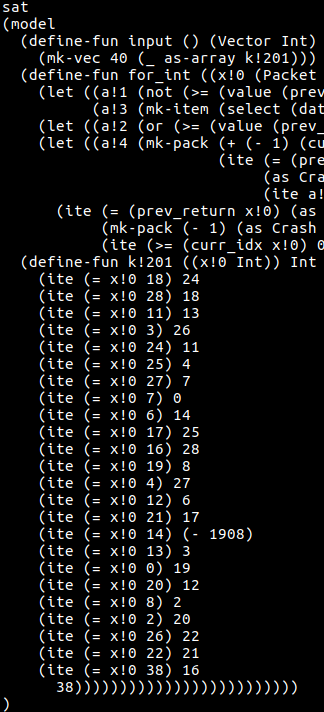
-- 90 lines omitted for clarity --



The resulting model identified the vector *[-1237, -1889, -1890, -1890, 1, 2, -5905]* as the input that would cause a crash. I attempted to test iterations of higher than 7 and found that for an iteration count of higher than 11, Z3 was unable to find a satisfiable model. Z3 would hang indefinitely (as much as I could tell) or respond with a message saying that it was unknown whether the assertions were satisfiable. Recursive functions are at the very limit of the theories that Z3 can handle. Additionally, my use of Array theory to create pseudo-lambda functions that could be passed in as arguments significantly increased the computation required, because Z3 needed to generate a definition for the *for\_body* constant (although *body\_statements* was an already a defined function). I preferred this modeled of a for-loop because *for\_int* was generic over any body since the body function could be passed in as a lambda (described earlier). However, a cap of 11 iterations was not satisfying. Thus, I decided to move in the direction of a less generic solution that could allow for more iterations.

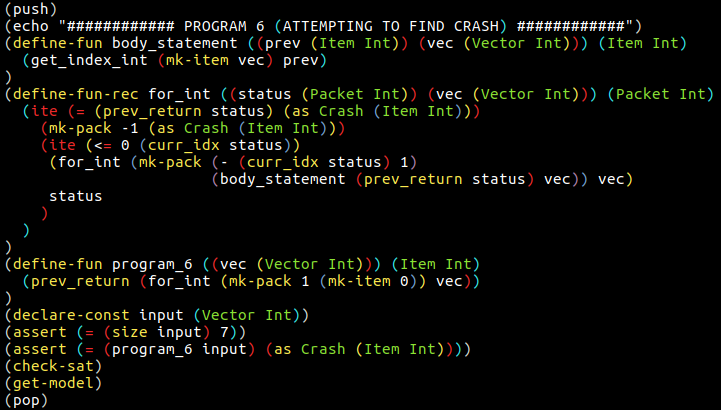


In this model, instead of passing in the body of the for-loop as an *Array*, the function *for\_int* directly calls the *body\_statement* function. The tradeoff, however, is that a new *for\_int* function must be modeled for each for-loop in the program. However, as evident by the result below, Z3 is able to create a model that satisfies all the assertions even at moderate iterations (40 for this test).

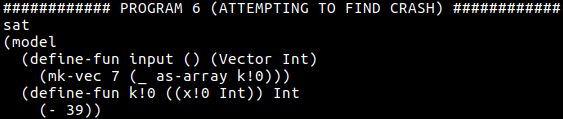


*Z3 output for Program\_5. Horizontal.*

To counteract the lack of generalness in the for-loop model in *program\_5*, I attempted to change the model for for-loops and for-bodies to take in a vector as an argument instead of directly accessing the *input* vector in *body\_statement*. However, this resulted in an iteration cap of 1 (mirroring the cap found in *program\_4)*. Possibly, this model approached the limits of the recursive function theory used by Z3.



*Program\_6 model.*



*Z3 output for Program\_6*

4 CONCLUSIONS

4.1 Summary

In summary, I developed a framework to model programs for the purposes of finding inputs that will cause them to crash. For simple programs containing only integer operations, array index retrievals, and if/else statements, Z3 proved the satisfiability of the assertions (in the program models) and generated models containing input vectors that would crash the programs. For-loops provided a greater challenge to model, since recursive function theories are at the limits of what Z3 can handle and higher-order function theories do not exist in SMT-LIB v2. Three implementations of for-loops provided different trade-offs. The first model of a for-loop in *program\_4* was generic over any body because it could take in a body function as an argument (by using Array theory to imitate higher-order functions and lambdas). However, Z3 could only handle up to 11 iterations. A less generic model was used in *program\_5* that succeeded to find crashing inputs on moderately high iteration counts (40 iterations) and vector sizes (40 elements). Finally, a third for-loop model was created that failed to reach more than 2 iterations.

4.2 Future work

Although this was an interesting approach to fuzzing, the limits of this approach were hit quickly. Despite its successes on simple programs, Z3 was unable to generate models that could satisfy more functionally complex programs. For example, very high iterations on for-loops with complex body statements failed to produce satisfying models. This is maybe why successful fuzzers use a mixture of randomized inputs and constraint solving to find crash causing inputs (also known as concolic execution). Randomized inputs are used to increase code coverage in areas where the constraints are lax and constraint solving for more efficiently reaching areas of code than randomly trying inputs.

REFERENCES

|  |  |
| --- | --- |
| [1] | “Concolic Testing.” Wikipedia, Wikimedia Foundation, 19 Sept. 2017, en.wikipedia.org/wiki/Concolic\_testing. |
| [2] | “Fuzzing.” Wikipedia, Wikimedia Foundation, 5 Dec. 2017, en.wikipedia.org/wiki/Fuzzing. |
| [3] | “Satisfiability modulo Theories.” Wikipedia, Wikimedia Foundation, 4 Dec. 2017, en.wikipedia.org/wiki/Satisfiability\_modulo\_theories. |
| [4] | Tinelli, Cesare. “Core.” The Satisfiability Modulo Theories Library, SMT-LIB, 25 Apr. 2015, smtlib.cs.uiowa.edu/theories-Core.shtml. |
| [5] | Tinelli, Cesare. “Ints.” The Satisfiability Modulo Theories Library, SMT-LIB, 25 Apr. 2015, smtlib.cs.uiowa.edu/theories-Ints.shtml. |
| [6] | Tinelli, Cesare. “ArraysEx.” The Satisfiability Modulo Theories Library, SMT-LIB, 25 Apr. 2015, smtlib.cs.uiowa.edu/theories-ArraysEx.shtml. |
| [6] | Cok, David R. “The SMT-LIBv2 Language and Tools: A Tutorial.” 23 Nov. 2013, smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf. |
| [6] | “Z3Prover/z3.” GitHub, 4 Dec. 2017, github.com/Z3Prover/z3. |
| [6] | Barrett, Clark, et al. “The SMT-LIB Standard Version 2.6.” 18 July 2017, smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf. “Z3Prover/z3.” GitHub, 4 Dec. 2017, github.com/Z3Prover/z3 |
| [6] | Barrett, Clark, et al. “The SMT-LIB Standard Version 2.6.” 18 July 2017, smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf. “Z3Prover/z3.” GitHub, 4 Dec. 2017, github.com/Z3Prover/z3. |

Journal Name:ACM Trans. Graph.

Volume:4

Issue:2

Year:2017

Month:October

Article Seq:XXXX

DOI:124564

Copyright Year:2017

Copyright Statement:rightsretained

ISSN:0730-0301

Price:$15.00