Extending CREST with multiple SMT solvers and real arithmetic

Do Quoc Huy, Truong Anh Hoang, Nguyen Ngoc Binh
University of Engineering and Technology, VNU
144 Xuan Thuy, Hanoi, Vietnam
{huydq.mcs07, hoangta, nnbinh}@vnu.edu.vn

Abstract

Generating the test inputs, that have high code coverage while minimizing the number of test inputs, is a practical but difficult problem. The application of symbolic execution in combination with SMT solvers gives a promising way to solve it. Recently, there have been several tools that help generating the test inputs for C programs, but their abilities are still limited, depending on the particular chosen SMT solver and most of them currently do not support real arithmetic. We propose an approach to overcome the limitation of unique solver's ability by using multiple SMT solvers and combining their results to get the best solution. We also propose a method of reasoning real arithmetic for symbolic testing. We have implemented this approach in an open source symbolic testing tool called realCREST. Our experimental results are very positive.

Keywords: Software Testing, Symbolic Execution, SMT solver, Test Inputs, C Programs

1. Introduction

Testing is the primary way to validate the correctness of software. Testing with manually generated inputs is the predominant technique in practice to ensure software quality. It accounts for 50-80% software development cost. Manual test inputs generation is expensive, error-prone, and rarely exhaustive. Thus, several techniques have been proposed to automate this task. They can be divided into two main techniques: random testing and symbolic execution.

Random testing [4, 8, 6, 17] is a simple technique for automated testing in which test inputs are generated randomly. A key advantage of random testing is that it scales well with random test input generation takes negligible time. However, random testing is extremely unlikely to test all possible behaviors of a program.

To generate test inputs that can explore as many branches of a program as possible, some techniques based on symbolic execution [13] have been proposed. Such techniques attempt to symbolically execute a program under test along with all possible execution paths of the program, generating and solving path constraints of program variables to follow these paths to produce concrete inputs that test them. Usually the constraints solving is delegated to some external SMT solvers [14, 15, 18]. Section 2 will explain this method in more detail.

In general, it is impossible to choose the most powerful solver because there are many of them such as Z3 [14], CVC3 [3], Yices [10], Barcelogic [15, 9] and they are different in their speed, memory use, and algorithms as well as underlying theories they support. A solver can be better than another in solving linear integer arithmetic constraints but it is worse when dealing with non-linear real arithmetic. In short, the set of problems that each solver can solve are not the same and their union is usually strictly larger than each of them. Test inputs generation tool usually uses only some particular SMT solvers or let users to choose one among several options. In practice, this is not convenient. Combining the power of these solvers allow us to make test generation tool much more powerful and convenient to use. This is one of the features that we implemented in our testing tool.

CREST [11] (http://code.google.com/p/crest/) is a tool used to generate test automatically for C programs. Using CIL [13], it inserts instrumentation code into a target program to perform symbolic execution to generate symbolic constraints along with the concrete execution. Then, it uses Yices to solve the generated symbolic constraints to get inputs that drive the test execution down to new, unexplored program paths. However, since CREST currently reasons symbolically only about linear, integer arithmetic, it cannot handle C programs that use real variables. In addition, because CREST uses only Yices to solve the path constraints, it sometimes cannot generate inputs for complex constraints that other solvers can do.

In order to overcome this problem, in this paper, we propose an approach to enhance the quality of solving constraints in generating test input problem by using multiple SMT solvers and combining their results to get the best so-

978-0-7695-4213-3/10 $25.00 © 2010 IEEE
DOI 10.1109/KSE.2010.34

lutions. This approach is implemented in a tool called realCREST, based on CREST architecture. The limitation in reasonable arithmetic of CREST is solved by revising instrumentation and symbolic execution modules to supply the ability of reasoning on both integer and real arithmetic. The method of using SMT solver and symbolic execution in generating testing, and the way we combine multi solvers to solve constraints are described in section 3. An overview of our tool and its improvement are showed in section 4. Section 5 presents experimental results and gives some discussions. At last,in section 6, we draw to close and point to our future works.

2 Related work

There are some tools to generate test input for C programs:

Directed Automated Random Testing (DART) [16], is an automated testing tool that combining three techniques to achieve high path coverage. First, it determines and simulates the "most general environment" in which a program can run by extracting the external interface of the program. Second, it creates a test driver which enforces again and again the program with irregular input. The execution of this input allows the tool to complete dynamical analysis on the track of the execution tree of the program. Eventually, DART can enforce the execution together with an alternative path by working out a relevant set of constraints and run concrete execution with the solution. It applies a solver for linear integer arithmetic. When the constraints are beyond this theory, the systems will be back to the concrete values of variables to go on with the execution.

By getting in group concrete execution with symbolic analysis, DART succeeds in solving the issue of false alerts sequences of the imprecision of the latter. Every inaccuracy reported by the tool is sonorous, because it depends on an actual execution of the program. The method proves to be useful and advantageous, especially with the use of library functions whose source code can not be analyzed. Whereas static analysis can not get anything about their use, DART can gain a lot of benefits from inspecting the real value back by such a function.

CUTE [12], an acronym for the Concolic Unit Testing Engine, is a tool built on the same work. Like DART, it also combines concrete and symbolic execution to examine alternative execution paths in order to disclose potential assertion violations. Unlike DART, it does not only use lp_solve for solving linear constraints, but also introduces optimizations. To do this, it first syntactically detects a new sub-constraint which is the negation of another, then eliminates common sub-constraints. It solves the constraints in an incremental way, only taking into account sub-constraints which depend on the newly introduced one. CUTE also deals with predicates that involve pointer variables by investigating the equivalence classes for them.

Although there are two methods in constructing input data structures, the former only uses function calls, and thus, suffers from the disadvantage of relying on these functions. Solving the class invariant for data structures should be the primary choice for this process. Since CUTE does not make use of exact pointer analysis, its precision is only subject to the predicates encountered so far in the program. The trade-offs between performance - within the extents of tractability- and higher path coverage could be discussed in the context of this aspect of the tool.

KLEE [5] is a symbolic execution tool which is capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. KLEE manages to get a handle on both, the path explosion problem and the "environment problem" . KLEE is different to other tools in optimizing the constraint set in order to speed up constraint satisfiability tests upon each "unsafe" access (viz., pointer dereferences, asserts, etc). KLEE is designed as an operating system for symbolic processes. Just like a "real" operating system picks up one among several competing processes for scheduling, KLEE's scheduler selects a symbolic process amongst many for symbolic execution. KLEE's scheduler's target is to run those symbolic processes that have possibilities to offer biggest improvements in coverage. Each symbolic-process has a current set of constraints on the environment, which must be met for the current path to be reached. If KLEE detects a violation in the constraint set, it triggers an error and generates a test case. KLEE's approach of reducing time cost of solving process is to optimize away the large parts of the state of each symbolic-process before the constraints are passed on to the solver.

Although each of DART, CUTE and KLEE concentrates on different targets in symbolic testing, all of them do not currently support symbolic floating point arithmetic. It is the main disadvantage in applying them to practical programs.

3. Using Multiple SMT Solvers and Symbolic Execution to Generate Test Inputs

When we perform symbolic execution for a program, we can get execution path. This execution path can be used to create new test inputs by using SMT solver to solve the path constraints. The general technique of using symbolic execution and SMT solver to generate test input are outlined in Figure 1.

We can see in Figure 1 that SMT solver plays an important role in generating test input. Naturally, a combination of two or more SMT solvers will give a better result compare with one SMT solver in solving constraints, because

```
Create initial test values;
Repeat
    Perform symbolic execution along with concrete execution;
    If unreached branch exists then
        Build a constraint system to reach that branch;
            Solve the constraint by SMT Solver;
        If solving is successful then
            Generate test values from concrete model;
        End if
    End if
Until (all branches are reached) or (solving is not successful);
```

Figure 1. Symbolic testing technique

some complex constraints that cannot be solved by a solver are solved by another solver and vice versa. We introduce a strategy for using multiple SMT solvers to solve the path constraints. Our strategy uses two or more SMT solvers to solve a constraint serially. Figure 2 describes our algorithm in combining multiple SMT solvers. Firstly, we choose a solver to solve the constraint. If it fails to solve the constraint, the next SMT solver will be used. This phase is repeated util the constraint is solved successfully or there is no unused SMT solver. If the constraint is solved successfully, the achieved concrete model is created.

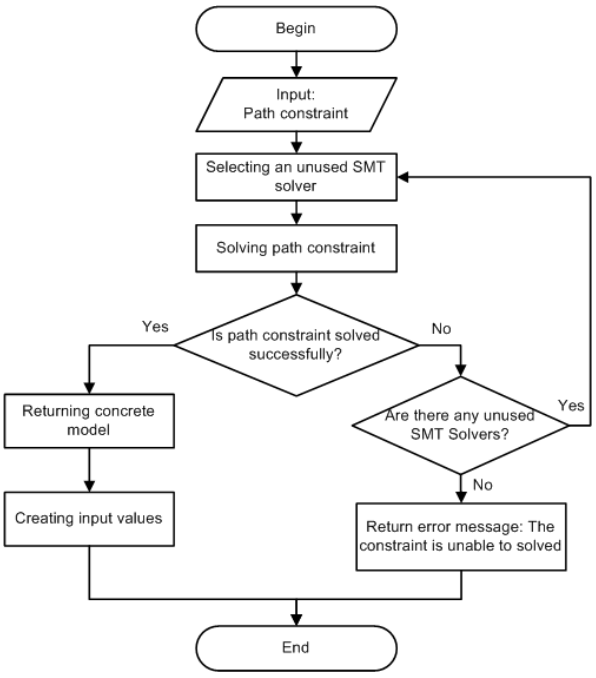

Figure 2. Combining multiple solvers algorithm

4. Test inputs generation for real numbers

As mentioned in the Introduction, CREST does not support generating test cases for real variables and since a large part of programs in practice uses real numbers, we extend CREST to deal with real variables in a tool called "real-CREST" . We have made the following important improvements compared to CREST:

- CREST can only perform symbolic execution with integral variables and solve only linear integer arithmetic constraints. realCREST can perform symbolic execution with both integral variables and real variables. The generated constraints which contain both linear integer arithmetic and linear real arithmetic are solved to produce test inputs for both integral and real variables.

- CREST can resolve only three arithmetic operators: addition, subtraction and multiplication. It can not resolve division operator. The reason is that it is able to solve only linear integer arithmetic, while the division operator may lead to real-type variables. In real-CREST, the division operator can be instrumented and performed in symbolic execution, like other operators.

- CREST uses only one SMT solver (Yices) to solve the path constraints, realCREST uses multiple SMT solvers so that more complex programs can be handled. Currently these SMT solvers are used sequentially. When a constraint cannot be solved by one SMT solver, another SMT solver is used. In the first version of realCREST, we use two solvers: Yices and CVC3. Yices is used firstly, and if it cannot solve a constraint, CVC3 will be used.

Architecture of realCREST is similar to CREST's architecture. It has three main modules like CREST, but we have made essential improvements in each module:

- Source code instrumentation module: This module instruments C source code, parsing it into conditional paths and stores this path to searching phrase. In this module, realCREST uses an extension file based on CIL to instrument source code and store execution paths into binary files. This file is an improvement of the original file of CREST to process real-type variables and division operator.

- C++ library for performing symbolic execution. This part includes classes that are used to run instrumented functions, catch and store program's state and execution path during symbolic execution process, and solve constraints to generate input for the next execution. Two classes in CREST: SymbolicExpr and SymbolicInterpreter are replaced by corresponding

classes SymbolicExprR, SymbolicInterpreterR in order to handle real-type variable's constraints and resolve division operator. Class YicesSolver of CREST is replaced by class Solver and two subclasses YicesSolver, CVC3Solver in order to combine two solvers in solving the constraint.

- Search strategies modules: This module supplies several different search algorithms to search unexplored path in execution path's space. realCREST reuses search strategies modules of CREST, with a small addition in class Search: one member variable (solver) is added in order to determine which solver is used to solve the constraint and invoke Solve method.

Operation schema of realCREST is depicted in Figure 3.

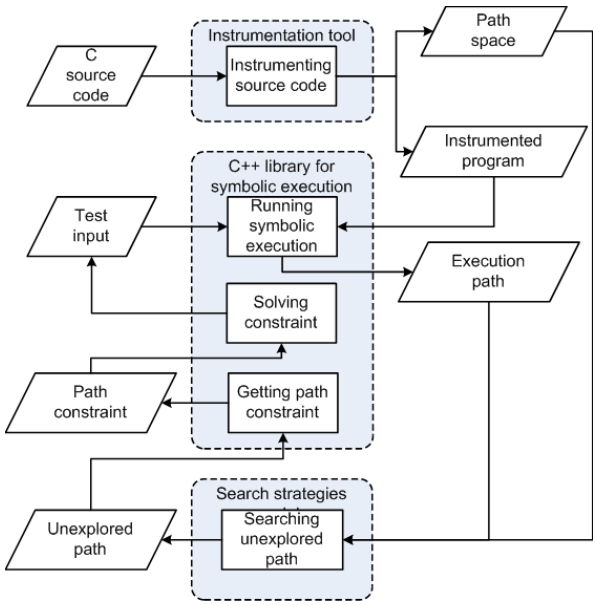

Figure 3. Overview of realCREST

5. Experiments and Discussions

We do experiments to test the extensions of realCREST and the efficiency of using multi solvers. The testing program is a simple C program which(used to) checks if three real numbers are length of three edges of a triangle, and what kind of triangle they construct. We insert some conditional statements to make the testing program become more complicated. The input variables are three float variables and one integer variable. This program can not be executed by CREST, because of the limitation of CREST in linear, integer arithmetic. We insert an include statement (#include "crest.h") and four corresponding macros to indicate them. The testing program is as below:

```
#include <stdio.h>
#include <crest.h>
int main(){
    float a,b,c;
    CREST_float(a);
    CREST_float(b);
    CREST_float(c);
    int x;
    CREST_int(x);
    if((a<=0)||(b<=0)||(c<=0))
      printf("is not a triangle\n");
    else if(((a+b)<=c)||((a+c)<=b)||((b+c)<=a))
      printf("is not a triangle\n");
    else if((a==b)&&(b==c))
      printf("is a equilateral triangle\n");
    else{
      if((2*a==b)&&(c/2>15))
        printf("is a special triangle\n");
      if((a>x)&&(x>10))
        printf("GOAL!")  ;
      if((a==b)||(b==c)||(a==c))
        printf("is a isosceles triangle\n");
      else
        printf("is a regular triangle\n");
    }
    return 0;
}
```

At first, testing program is instrumented by "crestc" batch file. The result shows that there are 30 branches, 36 nodes and 37 branches edges in testing program. Next, we run symbolic execution with instrumented code in three options: using only one solver Yices, using only one solver CVC3 and using multi solvers (Yices combining CVC3). The number of iterations we use in all cases is 1000. Table 1 shows the number of branches which have been covered when using different search strategies for each options, Figure 4 shows the time costs of each case with various search strategies.

Table 1. Branches coverage

Strategy	Yices	CVC3	Two solvers
Bounded DFS	29	28	**30**
CFG Base Line Search	25	22	27
CFG Heuristic Search	29	28	**30**
Hybrid Search	29	28	29
Random Input Search	25	25	25
Random Search	29	27	29
Uniform Random Search	29	28	**30**

In seven search strategies, Random Input Search can be bypassed, because it generates input randomly not based on solving constraints. Look at Table 1; it can be easily recognized that the best results are achieved when we use both two solvers to solve the constraints. All of branches are covered when three following strategies are chosen: Bound

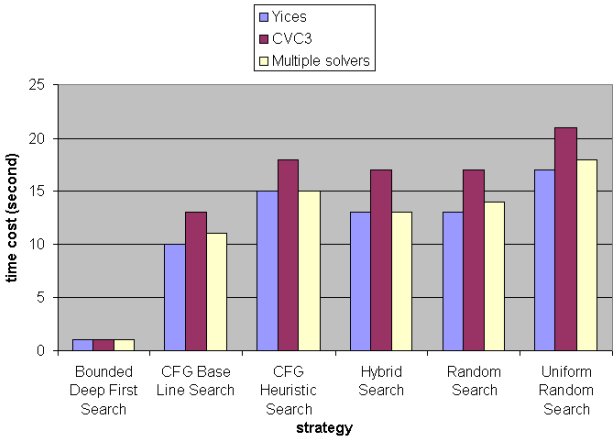

Figure 4. Time costs

Deep First Search, CFG Heuristic Search and Uniform Random Search. It means that they can explore all branches of the program. But if we use only one solver, the branches are not covered completely. It means that there are some path constraints which can be solved by one solver but can not be solved by another. With other strategies (CFG Base Line Search, Hybrid Search, Random Search), execution path space is not explored completely, therefore the coverage results are not stable, because they depend on what branches are reached by the search strategy. Therefore, we must repeat the experiment in many times and choose the highest result for each case. But in every case, the result of multiple solvers option is not worse than other options. The experiment result proves clearly that realCREST can resolve well linear arithmetic constraints (both integer and real) and division operator, especially the cooperating of two or more SMT solvers will make better result than using only one solver. Figure 4 proves that although multiple solvers bring lager code coverage than using only one solver, the time cost changes slightly. In most cases, the time cost of using multiple solvers is bigger than using Yices and smaller than using CVC3.

6 Conclusions and Future Work

We have instroduced realCREST which has several important contributions in expanding the class of solvable programs and raising the code coverage of generated test input in comparison with CREST. Its approach that uses multiple SMT solvers to solve the path constraints takes the full advantages of all solvers. The strategy of combining multiple SMT solvers that has been proposed is quite simple (in serial way) but the experiments give promising initial results.

In order to make realCREST become a more practical tool, there are some improvements we can do. First, we will make SMT solvers to execute in parallel to exploit the power of multicore CPUs or multiple CPUs system. Finally, we will develop realCREST to handle non-linear arithmetic, bit vectors and arrays, with the support of some SMT solvers that can solve these theories partially, such as Z3, CVC3, redlog.

References

[1] http://cil.sourceforge.net/.

[2] http://code.google.com/p/crest/.

[3] C. Barrett and C. Tinelli. Cvc3. *Lecture Notes in Computer Science*, 4590/2007:298–302, 2007.

[4] D. Bird and C. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22:229245, 1983.

[5] D. R. E. Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, 2008.

[6] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34:10251050, 2004.

[7] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for dpll(t). *CAV'2006*, 4144:81–94, 2006.

[8] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. *Proceedings of the 4th USENIX Windows System Symposium*, 2000.

[9] S. R. G. Necula, S. McPeak and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction,*, 2002.

[10] R. N. A. O. H. Ganzinger, G. Hagen and C. Tinelli. Dpll(t): Fast decision procedures. *Lect. Notes in Comp. Sci. (CAV'04)*, 3114:175–188, 2004.

[11] K. S. Jacob Burnim. Heuristics for scalable dynamic test generation. *No. UCB/EECS-2008-123*, 2008.

[12] D. M. K. Sen and G. Agha. Cute: A concolic unit testing engine for c. In *5th joint meeting of the European Software*.

[13] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385394, 1976.

[14] L. D. Mauro and N. Bjorn. Z3: An efficient smt solver. *Lecture Notes in Computer Science*, 4963:337–340, 2008.

[15] A. O. E. R.-C. Miquel Bofill, Robert Nieuwenhuis and A. Rubio. The barcelogic smt solver (tool paper). *Lecture Notes in Computer Science*, 5123:294–298, 2008.

[16] N. K. P. Godefroid and K. Sen. Dart: Directed automated random testing. *Proc. of the ACM SIGPLAN*, 2005.

[17] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *19th European Conference Object-Oriented Programming*, 2005.

[18] A. F. A. G. Roberto Bruttomesso, Alessandro Cimatti and R. Sebastiani. The mathsat 4 smt solver (tool paper). *Lecture Notes in Computer Science*, 5123:299–303, 2008.