Non-linear Arithmetic Support for CREST

Heechul Yun
Department of Computer Science
University of Illinois at Urbana and Champaign
heechul@illinois.edu

May 9, 2012

1 Introduction

Concolic testing tools, such as CUTE[1] and CREST[2], are very useful to find bugs especially for those are located in very rare execution paths. Those paths can only be reached when certain combination of inputs are supplied and therefore difficult to find. Concolic testing tools use SMT solvers to generate inputs that direct program to different execution path. However solvers are not complete in a sense that they may fail to solve some constraints. This limitation may result in reduced coverage of testing.

In this paper, we focus on CREST, an open source version of automated testing tool developed by Berkeley. CREST supports linear arithmetic constraints and it does not support non-linear arithmetic operations, division, and mod operators. Therefore if a constraint contains such unsupported operations, CREST may fail to find a path associated with the constraint. Such limitation is a combination of both (1) limitations of underlying SMT solver, YICES[3], and (2) limitations of CREST's internal design that assumes integer linear arithmetic constraint.

In order to increase coverage of testing, we extend CREST to support non-linear multiplication, division and modular operations. We achieve this by replacing underlying SMT solver to Z3[4] from Microsoft Research, and significantly restructuring the CREST core implementation especially related to its symbolic expression representation and the interaction with SMT solver. Our evaluation shows that our extension to CREST is capable of finding execution paths that depend on many complex non-linear operations.

The rest of this paper is organized as follows. In Section 2, we give general background of concolic testing and SMT solver. Section 3 describes limitation of CREST and investigate the reasons. Section 4 describes our extension. Section 5 gives evaluation results. We conclude in Section 6.

```
Step 1  Run the program with a given concrete inputs
Step 2  Collect symbolic path constraints.
Step 3  Pick a constraint and negate it.
Step 4  Solve constraints to generate new input
Step 5  If a solution is found, go to step 1 with the new
    input.
Step 6  If not found (or can't solve) goto step 3.
```

Figure 1: Concolic execution flowchart

2 Background

In this section, we first describe some basic information of symbolic execution and SMT solvers.

2.1 Concolic execution

Concolic execution is a combination of symbolic execution and concrete execution. Under Concolic testing, the program under testing is executed with concrete input values. When the program performs a statement that contain symbolic variable, it is symbolically executed and updated in a table together with a concrete value. Along the execution path, on each branch that uses at least one symbolic variable, the symbolic statement, called branch predicate, is stored for later use by the Concolic testing tool. Once the program finishes, the Concolic testing tool pick a predicate among the collected ones and negate it to enforce different execution path. The path predicates, of which the last one is negated, are then solved by a SMT solver, which we will describe in the next subsection, to generate concrete set of inputs that satisfy the given path constraints. A Concolic testing tool then execute the program under test with the newly generate input. This process continues until it execute all possible execution paths or limited by time. Figure 1 shows the overall flowchart of this process.

2.2 SMT solver

One crucial component in Concolic testing is SMT solver. An SMT solver determines satisfiability of a given set of formulas expressed in first-order logic. In Concolic testing, it is used to solve a satisfiability of a chosen set of path constraints. If the path constraints are solved, i.e., there exists a concrete value for each variable used in the predicates, the solution is used as inputs for the next iteration of the tested program. However, not all constraints can be solved because of both practical and theoretical reasons. Theoretically, some theories are essentially undecidable. One example is non-linear integer arithmetic. We say non-linear when the formula is expressed in a form of `(* a b)` where both a and b are variables (i.e., not numbers). Practically, there exists many SMT solvers

solver	supported theories
YICES	linear arithmetic, bitvector, array, uninterpreted function
Z3	linear arithmetic, non-linear arithmetic, bitvectors, array, uninterpreted function

Table 1: Comparison of SMT solver YICES and Z3.

but their supports are often very different in terms of quality and supported theories. Since the capability and performance of Concolic testing heavily depends on SMT solvers, it is worthwhile to survey existing SMT solvers.

We first compare YICES and Z3. YICES won several categories of SMT-COMP, an annual competition between SMT solvers, in 2009. Therefore it was a good choice when CREST was developed. Since then, however, the support and improvement seems to be lagging compared to competition. On the other hand Z3 from Microsoft is more recent development and provides more extensive support in terms of features and documentations. It also won several many areas of recent SMT solver competition held in 2011. Table 1 compares the two solvers. As shown in the table, YICES does not support non-linear arithmetic (e.g., `x * x`). Moreover, linear arithmetic support in YICES is weaker than Z3. For example, YICES does not support division and modular operation while Z3 does. Because these reasons, we choose to use Z3.

3 Limitation of CREST

In this section, we first describe type of programs that CREST may fail to find paths. We then describe the reasons of the problems.

```
int main(int argc, char *argv[])
{
        int x;

        CREST_int(x);

        if (x * x == 49) {
                printf("GOAL\n", x);
        }
        return 0;
}
```

Figure 2: Non-linear multiplication

```
int main(int argc, char *argv[])
{
        int x;

        CREST_int(x);

        if (x > 0) {
                if (x/2 == 2) {
                        printf("GOAL\n");
                }
        }
        return 0;
}
```

Figure 3: Division

```
int main(int argc, char *argv[])
{
        int x;

        CREST_int(x);

        if (x > 10 && x < 20) {
                if (x%10 == 3) {
                        printf("GOAL\n", x);
                }
        }
        return 0;
}
```

Figure 4: Mod operation

```
int main (int argc, char *argv[])
{
  int x;
  int A[] = { 0, 1, 2, 3, 4, 5 };
  CREST_int (x);
  if (A[x] == 2) {
      printf ("GOAL\n");
  }
  return 0;
}
```

Figure 5: Arrary

3.1 Examples of unsupported programs

We shows four examples that CREST fail to find a path of our interest (namely a path that prints "GOAL"). Figure 2 does not work under CREST because `x * x` is non-linear arithmetic. Figure 3 does not work because division operator (`/`) is not supported. Figure 4 does not work because mod operator (`%`) is not supported. Figure 5 does not work because array is not supported by CREST.

3.2 Limitation of SMT solver

The default SMT solver used in CREST is YICES from SRI. The YICES won several divisions of SMT solver competition in 2009 hence used in CREST. However it does not support non-linear arithmetic. Because of this, CREST cannot support cases like Figure 2. Also, YICES does not support common operators like division and mod. Therefore Figure 3 and Figure 4 are also not supported in CREST.

3.3 Limition of CREST implementation

In addition to limitations of YICES, CREST implementation itself has several issues that prevent from supporting anything but simple linear arithmetic operations. First, the core engine is designed to support only linear operations. Therefor even though we use a SMT solver that supports non-linear operations, its internal representation of symbolic expression cannot utilize them. Also, array is not supported because of its instrumentation although array theory is supported by YICES. Figure 5 shows an array example that CREST does not support.

4 Our Extension

In this section, we describe our extension to CREST. We first describe the type of programs that are supported by our extension. We then describe implementation details.

We extended CREST to support non-linear arithmetic, division, and modular operations. To do this, we changed underlying SMT solver to Z3 from Microsoft as it supports such features. However simply changing the part that accesses SMT solver was not enough because of the reasons mentioned in previous section. Therefore, we also significantly modified CREST's internal core classes in order to fully utilize Z3's capability.

CREST is implemented in C++ and has a sole class that is responsible to communicate with SMT solver, named `YiecesSolver`. Therefore, we initially thought that changing the class will be enough to support non-linear operations. It was, however, not enough because the other important classes that handle symbolic expressions are designed assuming linear operations. More precisely, classes like `SymbolicPred` and `SymbolicExpr` can only handle linear expressions.

```
src/Makefile                        |   12 +-
src/base/basic_types.h              |    2 +-
src/base/symbolic_execution.cc      |   39 ++-
src/base/symbolic_expression.cc     |  152 ++++++++++-
src/base/symbolic_expression.h      |    9 +
src/base/symbolic_interpreter.cc    |   41 +++-
src/base/symbolic_path.cc           |   77 +++++-
src/base/symbolic_predicate.cc      |   32 ++-
src/base/yices_solver.cc            |  576
    ++++++++++++++++++++++++++++++++++-
src/libcrest/crest.cc               |    2 +-
src/run_crest/concolic_search.cc    |    5 +-
11 files changed, 845 insertions(+), 102 deletions(-)
```

Figure 6: Diffstat of our implementation compared to original CREST

Test Pgm.	Orig.	Ours	Note
1	Pass	Pass	linear arithmetic
2	Pass	Pass	“
3	Pass	Pass	“
4	Pass	Pass	“
5	Fail	Pass	division
6	Fail	Pass	mod
7	Fail	Pass	non-linear arithmetic–`x*x`
8	Fail	Pass	“
9	Fail	Pass	`(x*x)%50`
10	Fail	Fail	array

Table 2: Evaluation result.

Therefore, we re-designed those classes to be able to express non-linear expressions. Figure 6 shows the modified portion of CREST for this project.

5 Evaluation

Our primary focus here is finding execution paths that the original CREST cannot find. We created a set of programs, totaling 10 examples, including examples presented in Section 3. Table 2 shows the result. Notice that our extension can find all examples that use a mix of linear and non-linear operations while the original CREST only can find paths when linear operations are used.

5.1 Limitation

Although supporting array is possible with using both YICES and Z3, it requires significant changes in the implementation of CREST. Therefore we left it as a future work.

6 Conclusion

In this project, we extended CREST to support non-linear arithmetic. We also support division and mod operations. It is achieved by using a recent SMT solver Z3 and with significant modification of CREST internal implementation. We evaluated our extension with several applications.

References

[1] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. ACM, 2005, pp. 263–272.

[2] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 2008, pp. 443–446.

[3] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, p. 2, 2006.

[4] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340.