Tool for analysis of Array Manipulating Programs using Rosette

Hitarth, Bhishmaraj

Introduction

Solver-aided programming language/framework, such as Rosette (Torlak and Bodik 2013) extend traditional programming languages with SAT/SMT-specific interface and constructs. Such a language framework makes it easier to embed/model domain-specific artifacts/systems and exploit use of SAT/SMT solver features (UNSAT, MAX-SAT, UNSAT-CORE, etc.) for performing various constraint-solving tasks, such as symbolic verification, debugging, bug localization, and synthesis.

Most of the current work in this field is focussed on arithmetic and bit-vector theories. There are tools for verification of programs in ANSI C with suitable assertions to a limited extent, like BugAssist (Jose and Majumdar 2011), but they don't focus on other solutions like synthesis. Also it's code is not open source.

In this project we propose to use Rosette-Racket for analysis ($verification$, $debugging$, and *bug-fixing*) of array manipulating programs. An array theory poses a challenge for symbolic analysis as it is undecidable, in general, because it requires quantifier instantiation. We simplify our problem by restricting ourselves to a decidable fragment of arrays theory (Christ and Hoenicke 2015) We simplify the problem of bug-fixing, which is essentially a synthesis problem, by restricting the grammar of the expressions that can be used in the fixes.

Problem: Automatic Verification, Debugging and Fixing Array Programs

Consider a simple program that is expected to swap the values at i and j index of an array a if they are not in ascending order.

We have deliberately introduced a **bug** in line 5 of the program by using j instead of i in the array select to preserve the post-assertion

```
1: int[10] a;
2: unsigned int i, j;
3: @Pre : assume(i < 10 && j < 10)
4: if (a[i] > a[j]) {
5:     temp = a[j]; //Bug!!
6:     a[i] = a[j];
7:     a[j] = temp; }
8: @Post : assert(a[i] ≤ a[j])
```

Our goal is to develop a prototype tool that does the following: (1) *verify* such program assertions, and if an assertion fails (2) localize *bugs* to a region (line 5, in this example) of the program, and suggest a possible fix (replace j by i) that makes the assertion true.

Approach

We will covert the program in logical formula using Racket/Rosette, and then proceed to the verification. Assume that the final formula we get is P.

Verification: We expect that $\neg P$ will be $UNSAT$. If it is the case, the program is verified.

Bug Localization: If $\neg P$ is SAT, then there is a bug in the program. We shall get one or more counter models for this SAT instance; for these models P must be UNSAT. We use the $UNSAT$ core for UNSAT proof of P under this model to filter out as much of the code which is not relevant for the bug. These tasks will be done using interface in Rosette for computing MAXSAT and UNSAT cores.

Bug-fixing: For the sake of simplicity, **we shall assume that the bug lies in the array access operations** in the program. For synthesis, we shall convert the program to a sketch by introducing *holes* in the array access operations, i.e., in the index expressions used in array select or update. Then, with the help of Rosette, we shall try to find out the possible substitution for the holes so that the $\neg P$ becomes $UNSAT$, and hence the program becomes correct.

Methodology

1. Describing a language which can be used to specify the array manipulating programs with the pre/post conditions and loop invariants
2. We will be implementing the tool in Rosette. The steps will be as follows:

- Embed the array programming language within Rosette
- Convert the program and assertions into SMT array logic in SMT lib format and check assertions.
- Use counter models and related UNSAT cores to perform bug localization.
- Synthesize bug fixes by performing syntax-guided search on the defined grammar for possible fixes.

3. Implementation of our method within the Rosette-Racket solver-aided programming tool/language framework.
4. Experiment our implementation on a targeted class of benchmark examples.

Detailed Approach

1. The grammar that we are allowing for the program is described below:

$$
\begin{array}{lll}
\langle \textit{Prog} \rangle & := & \langle \textit{pre-cond} \rangle \\
 & & \langle \textit{declaration} \rangle * \\
 & & \langle \textit{stmt-list} \rangle \\
 & & \langle \textit{post-cond} \rangle ; \\
 & & \\
\langle \textit{pre-cond} \rangle & ::= & \text{@}\mathbf{pre}(\langle \textit{assertion} \rangle); \\
 & & \\
\langle \textit{post-cond} \rangle & ::= & \text{@}\mathbf{post}(\langle \textit{assertion} \rangle); \\
 & & \\
\langle \textit{assertion} \rangle & ::= & (\langle \textit{assertion} \rangle \ \&\& \ \langle \textit{assertion} \rangle) \\
 & | & (\langle \textit{assertion} \rangle ::= \langle \textit{assertion} \rangle \ || \ \langle \textit{assertion} \rangle) \\
 & | & (\langle \textit{expr} \rangle = \langle \textit{expr} \rangle) \\
 & | & (\langle \textit{expr} \rangle \leq \langle \textit{expr} \rangle) \\
 & & \\
\langle \textit{declaration} \rangle & ::= & \mathbf{int} \ \langle \textit{int-var} \rangle ; \\
 & | & \mathbf{int[]} \ \langle \textit{array-var} \rangle ; \\
 & & \\
\langle \textit{assignment} \rangle & ::= & \langle \textit{ident} \rangle = \langle \textit{expr} \rangle ; \\
 & & \\
\langle \textit{stmt} \rangle & ::= & \langle \textit{assignment} \rangle \\
 & & \\
 & | & \mathbf{for} \ \mathbf{(}\langle \textit{ident} \rangle = \langle \textit{expr} \rangle \ \mathbf{to} \ \langle \textit{expr} \rangle\mathbf{)} \ \mathbf{do} \ \{ \\
 & & \langle \textit{pre-cond} \rangle \\
 & & \langle \textit{assignment} \rangle * \\
 & & \langle \textit{post-cond} \rangle \\
 & & \} \\
 & & \\
 & | & \mathbf{if} \ (\langle \textit{cond} \rangle) \ \mathbf{then} \ \mathbf{do} \ \{ \ \langle \textit{assignment} \rangle * \ \} \\
 & & \\
\langle \textit{expr} \rangle & ::= & \langle \textit{expr} \rangle + \langle \textit{expr} \rangle \\
 & | & \langle \textit{array-var} \rangle [\langle \textit{expr} \rangle]; \\
 & | & \langle \textit{int-var} \rangle \\
 & & \\
\langle \textit{stmt-list} \rangle & ::= & \langle \textit{statement} \rangle \ \mathbf{;} \ \langle \textit{stmt-list} \rangle \\
 & | & \langle \textit{stmt} \rangle ; \\
 & | & \langle \textit{empty} \rangle \\
 & & \\
\langle \textit{ident} \rangle & ::= & \langle \textit{integer-identifier} \rangle
\end{array}
$$

2. We will explain further how we can convert a general program written in this way to the Racket/Rosette Program.
3. Explore the **vector** datatype in Rosette and how to use it for our purposes.
4. Each loop will be verified separately using it's **pre** and **post** conditions only. If the loop passes the test, then in the final checking of program, the **post** condition of the loop will be assumed without verifiying the loop again.
5. ...

Expected Results

This is a work under progress being done as a project for an SMT course. By the time of SMT School we expect to have an implementation of the tool in Rosette with results of running our tool on a set of benchmark array manipulating programs.

References

Christ, J., and Hoenicke, J. 2015. Weakly equivalent arrays. In *International Symposium on Frontiers of Combining Systems*, 119–134. Springer.

Jose, M., and Majumdar, R. 2011. Cause clue clauses: error localization using maximum satisfiability. In *ACM SIGPLAN Notices*, volume 46, 437–446. ACM.

Torlak, E., and Bodik, R. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 135–152. ACM.