SASS Project Proposal : Trace Abstraction Framework

Kush Grover Arijit Shaw

November 8, 2018

1 Introduction

Software Model Checking has always been a interesting and challenging field. A modern and promising approach in this is Trace Abstraction.

Matthias et. al. started an approach [HHP13] of using automata in this field. Briefly it is an CEGAR based model checking strategy, where they used automata to generate assertion specific abstraction. Abstractions are generated using interpolants found using a theorem prover, which are in form of *Hoare triples* and the abstracted program is represented in a *Floyd Hoare Automaton*.

Goal : Implemnting a complete (with some limitations, which we have discussed later) Software Model Checking tool, that uses the approach taken by [HHP13].

2 Overview of Abstraction Algorithm

From an algorithmic point of view, the approach looks like following:

1. **Generate Trace :** From a given program automaton, find an accepting path in the automaton. If this is unsatisfiable, The sequence of statements on this path is infeasible (it does not have a possible execution). If this trace is satisfiable, then this path is a feasible trace which means that program is incorrect.

2. **Generate Interpolants :** Infeasible trace I is a step towards proving correctness of the program. To construct the correctness proof, we want to abstract the reason of infeasibility. To do this, we intend to get interpolants between program statements.

3. **Construct Floyd-Hoare Automaton :** To abstract the trace, we make a *Floyd-Hoare Automaton* from the trace, where the states represent the interpolants found from the trace. Between two states P and Q, we include all possible statements (alphabet of automaton) st, where $\{P\}st\{Q\}$ is a valid *Hoare triple*.

4. **Cover Check - Automaton :** One most important observations in [HHP13] is, $infeasibility \Rightarrow correctness$. Which in terms say, if FHA

$A_1, A_2, \ldots A_n$ cover all control-flow traces, then the program is correct. So, to prove the correctness of a program P, we have to show $P \subset A_1 \cup A_2 \cup \cdots \cup A_n$

5. To generate other reasons of infeasibilities we take $P \cap {A_i}^c$ as our current program automaton P, and start from step 1.

3 Implementation Plans

3.1 Input and Output

Input : Our input is a program in program automaton as `.ats` format, which is required by *AutomataLibrary*.

The states in automaton will represent program locations. The transitions of the automaton will be the program statements. The expressions are needed to be written using the following grammar.

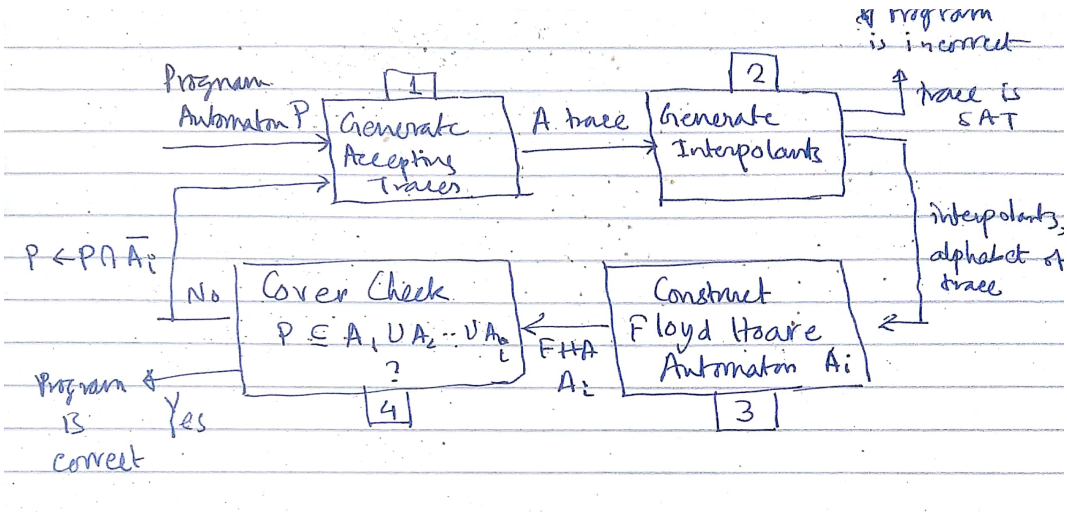
```
<expression> ::= <logical-or-expression>

<logical-or-expression> ::= <logical-and-expression>
                          | <logical-or-expression> ||
                              <logical-and-expression>

<logical-and-expression> ::= <inclusive-or-expression>
                           | <logical-and-expression> &&
                               <inclusive-or-expression>

<inclusive-or-expression> ::= <exclusive-or-expression>
                            | <inclusive-or-expression> |
                                <exclusive-or-expression>

<exclusive-or-expression> ::= <and-expression>
                            | <exclusive-or-expression> ^ <and-expression>

<and-expression> ::= <equality-expression>
                   | <and-expression> & <equality-expression>

<equality-expression> ::= <relational-expression>
                        | <equality-expression> == <relational-expression>
                        | <equality-expression> != <relational-expression>

<relational-expression> ::= <shift-expression>
                          | <relational-expression> < <shift-expression>
                          | <relational-expression> > <shift-expression>
                          | <relational-expression> <= <shift-expression>
                          | <relational-expression> >= <shift-expression>

<shift-expression> ::= <additive-expression>
                     | <shift-expression> << <additive-expression>
                     | <shift-expression> >> <additive-expression>

<additive-expression> ::= <multiplicative-expression>
```

```
                        | <additive-expression> +
                            <multiplicative-expression>
                        | <additive-expression> -
                            <multiplicative-expression>

<multiplicative-expression> ::= <primary-expression>
                              | <multiplicative-expression> *
                                  <primary-expression>
                              | <multiplicative-expression> /
                                  <primary-expression>

<primary-expression> ::= <identifier>
                       | <constant>
                       | ( <expression> )

<constant> ::= <integer-constant>
             | <floating-constant>

<expression> ::= <assignment-expression>

<assignment-expression> ::= <primary-expression> <assignment-operator>
    <assignment-expression>

<assignment-operator> ::= =
```

If program has data types other than `int` (e.g. `float` or `unsigned char` for bitvectors), then we need that to be specify it in another file.

Output : Result of verification - Success or Failure.

3.2 Flow of Algorithm



3.3 Implementation

- We plan to use *ULTIMATE Automata Library* [aut] for generating traces from program automata, check whether language of an automaton is subset of another, and check their emptiness.
- To generate interpolants from a trace, we use MathSAT. Inside our code, we used the MathSAT API, and passed the trace to it.
- To check validity of *Hoare triples*, we use the same SMT-solver.
- With these interpolants and valid Hoare triples, we made the Floyd-Hoare automata A_i and took the difference $P \setminus A_i$, where P was the program automaton. This $P \setminus A_i$ is our program automaton for next iteration.

3.4 Limitations

- Works only for Linear Integer Arithmetic and Bitvectors.
- Input must be given in specified format.
- Program must not contain function calls.

3.5 Bottlenecks

- Time taken by Automata Library is too much.
- Time taken to check validity of Hoare triples. For smaller programs too, we are checking a large number of Hoare triples.

3.6 Optimization

- **Choose hoare triples to check smartly.**
- **Manually inserting invariants.**

4 Current Status and Division of Labour

The program has been implemented and a working version is available at `https://github.com/kushgrover/ApnaAutomizer/`.

We believe that division of labour creates less-skilled workers and wanted to skip that from our work. For technical necessities, we did something like this. Kush was involved in steps 2 and 3. Arijit was involved in steps 1, 2 and 4. Both were involved in testing and working on examples.

4.1 Experiments

We intend to test on some programs from literature which are part of *SV-COMP* benchmarks. Those are available at `https://github.com/charles729/examples`.

The current version was successful in verifing a few programs. The test programs include some benchmark programs along with loops, branching and nested loops.

References

[aut] https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automata_library.

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *International Conference on Computer Aided Verification*, pages 36–52. Springer, 2013.