

# Array Program Verifier

A tool for verification and bug localization for simple program involving arrays

## Content:

- Background
  - Verification
  - Bug Localization
- The structure of the tool
- Examples
- Conclusion

## By:

- Hitarth
- Bhishmaraj

# Verification.

- Program verification involves proving that the program satisfies the given requirements.
- We achieve that by carefully converting the program to logical assertions, and encoding the provided pre and post conditions.
- Let the pre, post condition and the program as logical assertions be  $P$ ,  $Q$  and  $S$ , respectively. We form the Hoare Tuple  $\{P\} S \{Q\}$ . Now, the program is valid if and only if  $P \wedge S \wedge \neg Q$  is **unsatisfiable**.
- Given the program in our predefined grammar, we convert the program into set of **z3** assertions while converting it to Static Single Assignment form.
- We get the precondition  $P$ , postcondition  $Q$  and the program as  $S$ . Then we perform the verification as described earlier.

# Bug Localization.

- Bug localization involves finding the possible statement in the program which might be the cause of the error when the program verification fails.
- It cannot be guaranteed to be always correct.
- We, in this tool, use **Minimal Unsat Core** to localize the bug.
- Given the Hoare Tuple  $\{P\} S \{Q\}$  for the program, as the program is incorrect,  $P \wedge S \wedge \sim Q$  would be SAT. We find the model  $\mathcal{M}$  for the assertion  $P \wedge S \wedge \sim Q$ . Notice that the program fails under this model.
- Under the model  $\mathcal{M}$ , the assertion  $P \wedge S \wedge Q$  would be **UNSAT**. We find the MUC for the assertion  $P \wedge S \wedge Q \wedge M$ , where  $M$  is the conjunction of clauses as per the values assigned by the model  $\mathcal{M}$ .
- We find the lines in the program associated with the clauses contained in this MUC, and return them as the possible cause of error in the program.

# Structure of the Tool

- The tool can be decomposed into four modules:
  1. **Parser:** Reads the program and converts it to a tree for further processing. We have used **Lark (Python Library)** for that.
  2. **Encoding as Z3 constraints:** In this module, given the parsed tree of the program, we get the precondition, postcondition and program in logical form as Z3 constraints (while converting the program to SSA).
  3. **Verifier:** Given the precondition, postcondition and the program execution encoded as Z3 constraints, we proceed to verify the program by calling the solver on the constraints as described earlier.
  4. **Bug Localizer:** If the program verification fails in the 3<sup>rd</sup> module, then this module tries to find the unsat core of the constraint system, as described previously, to localize the possible statements cause of the error.

## Structure of the Tool (Contd.)

The grammar of the program (broadly) is described below:

$\langle Program \rangle$	$::=$ <b>@pre</b> ( $\langle assertion \rangle$ ); $\langle declaration \rangle^*$ $\langle stmt \rangle^*$ <b>@post</b> ( $\langle assertion \rangle$ );
$\langle declaration \rangle$	$::=$ <b>int</b> $\langle int-var \rangle$ ;   <b>int</b> [] $\langle array-var \rangle$ ;
$\langle stmt \rangle$	$::=$ $\langle assignment \rangle$   <b>if</b> ( $\langle cond \rangle$ ) <b>then do</b> { $\langle assignment \rangle^*$ } <b>else</b> $\langle assignment \rangle^*$
$\langle assignment \rangle$	$::=$ $\langle ident \rangle = \langle expr \rangle$ ; $\langle ident \rangle$ $::=$ $\langle integer-identifier \rangle$   $\langle array-identifier \rangle[\langle integer-identifier \rangle]$   $\langle array-identifier \rangle[\langle number \rangle]$

# Examples.

## 1. Simple Sum

- We pass the following program for verification to our tool:

```
1 @pre(((i<10 and j<10) and a[i] > a[j]));
2 int i;
3 int j;
4 int x;
5 int y;
6 int[] a;
7 a[i] = x;
8 a[j] = y;
9 a[0] = (a[i] + a[j]);
10 @post(a[0] = (x + y));
```

- The tool identifies the program is valid!

```
(base) Hit-Laptop:Project hitarth$ python atpy.py at_example_1.atlang
Program is syntactically correct!
-----
The program is valid!
(base) Hit-Laptop:Project hitarth$
```

## Examples.

### 2. Simple Sum (Buggy)

- We pass the following program for verification to our tool:

```
1 @pre(((i<10 and j<10) and a[i] > a[j]));
2 int i;
3 int j;
4 int x;
5 int y;
6 int[] a;
7 a[i] = x;
8 a[j] = y;
9 a[0] = (a[i] + a[i]); #Bug
10 @post(a[0] = (x + y));
```

- The tool identifies the program is **invalid** and also identifies the buggy line!!

```
Following are possible reason for the error:
    Line #9: a[0] = (a[i] + a[i]); #Bug
(base) Hit-Laptop:Project hitarth$
```

# Examples.

## 2. Swap Program

- On running this program, our tool **correctly** identifies that the program is not correct. But it outputs the bug to be due to the line #9, which is **not** correctly identified.

```
(base) Hit-Laptop:Project hitarth$ python atpy.py at_src_swap.atlang
Program is syntactically correct!
sat
The program is invalid!
Following SAT model is found:
a_0          - Store(Store(K(Int, 6), 1, 8855), 0, 16575)
stmt_rhs_9   - 8855
p1           - True
a_2          - Store(Store(K(Int, 6), 1, 8855), 0, 8855)
p3           - True
a_1          - Store(Store(K(Int, 6), 1, 8855), 0, 8855)
precond      - True
i_0          - 0
p4           - True
p2           - True
stmt_rhs_7   - 8855
postcond     - True
j_0          - 1
stmt_rhs_8   - 8855
temp_1       - 8855
p5           - True
p0           - True

Following are possible reason for the error:
Line #9: a[j] = temp;
(base) Hit-Laptop:Project hitarth$
```



# Conclusions.

- The tool is able to verify the program correctly.
- It cannot always localize the bug correctly, and sometimes can't find the bug at all.
- The tool presently doesn't supports loops. We shall try to include that in future.
- We can make the bug localization better by trying MAXSAT instead of MUC as sometimes Z<sub>3</sub> cannot find the unsat core.



The End.

