Writing a Software Reliability Tool

Andrei Antonescu, Irina Veliche, Yangfan Zhang *

Imperial College London
aba111, iv511, yz10111@doc.ic.ac.uk

Abstract

"Reliable software must include extra, often redundant code to perform the necessary checking for exceptional conditions. This reduces program execution speed and increases the amount of store required by the program. Reliability should always take precedence over efficiency for the following reasons: computers are now cheap and fast, unreliable software is liable to be discarded by users, system failure costs may be enormous, unreliable systems are difficult to improve and may cause information loss." [1]

I. Introduction

For this coursework, we implemented a *SRTool*, to analyse single-procedure *Simple C* programs. We applied standard and precise techniques for non-loop code like *Predication*. To analyze loops there isn't a perfect technique so we used both under-aproximation techniques like *BMC* and over-aproximation techniques like *Loop Summarisation* combined with *Invariant Generation* and *Houdini*. For the competition part we applied different heuristics to enrich the default techniques together with compiler optimizations to reduce the workload of the SMT solver where possible. The pinnacle of tool's reliability came when combining techniques so they complement their weaknesses. In the following sections we will discuss in detail the implementation to enlight the challanges we faced and the insight we got.

*Thanks to Alastair Donaldson, Paul Thomson

[1] `https://sites.google.com/site/assignmentssolved/mca/semester3/mc0071/10`

II. Turning an SSA-form program into an SMT-LIB formula

II.1 Transform each operator into SMT-LIB

We implemented each type of *BinaryExpr* and *UnaryExpr* as descibed in the SMT-LIB documentation [2]. We converted each expression that returns a boolean to bit vector using *tobv32* and we defined an auxilary function *tobool* to convert each argument to a logical expression from bit vector to boolean. For simplicity we allowed an *IntLiteral* to hold a negative value so we added a special case that would generate a minus *UnaryExpr* with the absolute value. We took special care to implement *RSHIFT* via *bvashr* and *MOD* via *bvsdiv* to preserve the sign of the number. To make sure our implementation is correct we created a testcase that contains all possible operators.

II.2 Build the query

This part was quite straightforward as we just declared all variables and create new boolean properties to hold the values of program's assertions. After that we transformed all assignments into assertions and negated all original assertions. The latter was required since we want the SMT solver to find an assignment that would break at least one assertion instead of finding an asignment that respects all of them. To achieve this we added a disjunction between all negated assertions as this evaluates to true if at least one fails, hence the solver returns *sat*. Notice that this is the only *real* assertion in our program because all the original ones assign their truth value to a variable. To better illustrate our design we added an example:

Original statements

```
1. Assignment
        x = expr

2. Assert
        assert(expr)
```

Query translation

```
1. Assignment
        assert(= x expr)

2. Assert
        assert(= prop_x (lnot expr))

3. Final Assert
        assert(prop_1 ∨ prop_2 ∨ ... prop_n)
```

II.3 Getting failed assertions

After implementing BMC and Houdini we noticed we need to keep track of *UnwindingAssertions* and *HoudiniAssertions* to be able to check when the program is *sat* if it is because of a loop reaching its unwind limit or because of a failed candidate invariant.

III. Single static assignment

In SMT-LIB assignments are implemented via asserts so it won't be possible to assign to the same variable multiple times. Because of that we implemented a SSA visitor that creates a new variable for each *not new* assignment or if the assignment references the variables beeing assigned to $(x = x + 1)$. It was pretty straightforward to implement it using a HashSet to keep track of *seen* variables and a HashMap to keep track of current index for a variable. We extended the *DeclRef* class to keep track of index as well so we can change the naming format easily $(x- > x\$1)$.

[2] http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2

IV. Predication

Predication represents the skeleton of transforming a *Simple C* program into a SMT-LIB logic formula. We implemented each statement along the lines discussed in the lecture. We used two types of predicates to model behaviour of conditionals and assume statements. The predicate for *IfStmt* was always depending on the parent predicate and for *AssumeStmt* it was global, hence only depending on the previous one. Next we will show how we predicated each type of statement. For simplicity let's define P_{cond} to represent the top predicate for conditionals (the variables that keep the result of evaluating last *IfStmt* condition) and P_{assume} to represent the top predicate for assume statements (as above for *AssumeStmt*).

Original statements

```
1. Assume
     assume(expr)

2. Assert
     assert(expr)

3. Assignment
     x = expr
4. Havoc
     havoc(x)

5. Conditional
     if (expr)
         thenStmts
     else
         elseStmts
```

SMT-LIB ready statements

1. Assume
   $P_{assume} = ((P_{cond} \implies expr) \wedge P_{assume})$

2. Assert
   $assert((P_{cond} \wedge P_{assume}) \implies expr)$

3. Assignment
   $x = ((P_{cond} \wedge P_{assume})\ \texttt{?}\ expr : x)$

4. Havoc
   `int h1`
   $x = ((P_{cond} \wedge P_{assume})\ \texttt{?}\ h1 : x)$

5. Conditional
   $P2_{cond} = (P_{cond} \wedge expr)$
   `# top` P_{cond} `predicate becomes P2`
   `thenStmts`

   $P3_{cond} = (P_{cond} \wedge \neg expr)$
   `# top` P_{cond} `predicate becomes P3`
   `elseStmts`

V. Bounded model checking mode

Because a logic formula cannot model a loop, one way of analyzing the program is to unwind loops up to a certain depth. This is an under-approximation of the program for which we can protect against false positives by adding unwinding assertions. The disadvantage is that it won't discover bugs after a certain number of iterations and therefore it cannot guarantee that a program is correct if the loop can exceed the unwinding depth. We implemented loop unwinding as described in the lectures and applied a few tricks that we will explain next.

V.1 Custom loop unwinding

We discovered that a fixed unwinding depth is very bad in practice because either it timeouts on nested loops or cannot check even $\log_2 n$ iterations. Because of that we keep track for each loop how many loops it contains inside its loop body and how many contain it. Depending on the lenght of the loop chain we unwinded loops such that we have around 30 to 60 total unwinds per block of nested loops.

V.2 Infinite loop detection

Infinite loops can be replaced with assume(false) as long as they don't contain assertions. This won't change the behaviour of the program and will win valuable time for the SMT solver. We used a few heuristics to check for infinte loops, the main one beeing if the variables referenced in the loop condition are not part of the loop's modset.

VI. Verifier

Loop summarisation is a very powerful technique to analyze the behaviour of a loop as long good invariants are provided. This techinque represents an over-aproximation of the program so it can expose more behaviour than in the original program and result in false positives. After we combined the tool with Houdini and Invariant Generation it became quite powerful.

VI.1 Asserts in the loop body

Straightforward loop sumarisation cannot be applied on loops that contain assertions. However, assertions that reference variables that are not modified from loop head to statement or from statement to loop head will still respect the invariants so we can abstract them as well. To handle this case we generated an extra if statement with the loop condition:

```
if (loop condition)
{
    havoc(∀x ∈ modeset_loop)
    assume(∀inv ∈ invariants_loop)
    assert(assertions_loop)
}
havoc(∀x ∈ modeset_loop)
assume(∀inv ∈ invariants_loop)
assume(not condition_loop)
```

We checked where there are assertions in the loop using a normal visitor combined with a reverse visitor. If we had assertions that referenced variables modified since the loop head and that are modified again before the end of the loop we would fail the loop summarisation:

```
while (condition)
{
    x = x + a
    assert(x)
    x = x + b
}
```

VII. Houdini

VII.1 Houdini Transformer

We implemented Houdini in two steps. In the first phase we visit each *WhileStmt* and transform it in a *Houdini* block by performing loop abstraction as described below. We extended the *BlockStmt* class to keep track of the original *WhileStmt* so we can put it back in the program once the failing candidate invariants are removed. For the sake of clarity we show here how a *WhileStmt* is transformed after the algorithm runs:

Original loop

```
while (condition_loop)
  candidate(invariant_i)
  invariant(invariant_i)
{
    # modifies ∀x ∈ modset_loop
    loopBody;
}
```

Abstracted Houdini loop

```
assert(∀inv ∈ invariants_loop)
havoc(∀x ∈ modset_loop)
assume(∀inv ∈ invariants_loop)
if (condition_loop)
{
    loopBody;
    assert(∀inv ∈ invariants_loop);
    assume(false);
}
```

VII.2 Houdini Verifier

In the second phase we visit each *Houdini Block* and remove all failing invariants. In the end after the SMT solver returns *unsat* we conclude that the ones left are correct and return the original *WhileStmt* with the correct candidades turned into normal invariants. Implementation-wise, we removed candidates in a loop that calls the SMT solver on the program and returns failing invariants by examining failed Houdini assertions. Because we want only Houdini assertions from the current block to fail we carefully pick those when we send the query to the solver.

One technical difficulty we encountered was to replace a node with a new one in the program without returning the latter from the visit function. To achieve that we kept track of the parent node and child index in the *DefaultVisitor* and created a function that replaces the child reference for the current node live. We designed the code this way for efficiency to remove all failed invariants in one go without the need of running the visitor multiple times. We were impressed with the overall performance of the algorithm, partially justified because we removed usually more than one invariant per call to SMT solver and we removed assertions unrelated to the current block, subsequently reducing the search space. On a benchmark the algorithm was able to check 47 invariants per second. This result allowed us to use a more aggressive approach in Invariant Generation.

VIII. Candidate Invariant Generation

We constructed candidate invariants by placing various variables or constants on both sides of each comparison operator in the Simple C grammar. After inspecting the sample test cases, we chose the variables and constants using the following four heuristic methods.

VIII.1 Comparing modified and used variables in loops

Each while loop contains statements that use and modify some variables. We constructed our first set of invariants by comparing each modified variable with each used variable, using every comparison operator available.

VIII.2 Comparing modified variables in loops and constants

Some valid invariants are established by comparing the modified variables with some constants. In addition to the common boundary testing constants such as 0, 1, and -1, we selected all constants used in the assertion statements in the entire program. This way, we included all constants used in assertions before and after the loops.

VIII.3 Comparing variables used in assertions

We next built candidate invariants that compare variables used in all assertion statements in the program, as these variables may have relationships that make them true invariants. Here, we chose comparison operators selectively so that we do not pass repeated cases such as $x < y$ and $y > x$ for evaluation in Houdini mode.

VIII.4 Checking parity of modified variables

Lastly, we created candidate invariants that checked the parity of each modified variable. This intuition is derived from inspecting the test case *tests/invgen/test_down2_good.sc* where the modified variable i stays as an even number before and after each loop. Adding the candidate invariant $(i\%2) == 0$ was crucial to passing this test case.

IX. Competition

IX.1 Adding testcases

We liked the competition part a lot because it was cool to explore different verification tehniques in a simple language without getting stuck in implementation details but still very difficult to verify programs for correctness. First we started adding more testcases because the program could pass all of the provided ones. We added more than 40 testcases in total. From simple algorithms like: *mergesort, gcd, fibonaci, parity* to more complex ones like *modulus division, factorisation, Fermant's last theorem, Miller-Rabin primality testing, Charmichael numbers and Smith numbers*. Some of the programs were difficult to write without arrays so we used big if statements to simulate arrays access.

IX.2 Compiler optimisations

We decided to apply standard compiler optimisations to the AST program like constant folding and dead code elimination. We implemented constant folding by extending a visitor that keeps a state of variables if they are assigned to *IntLiterals*. Each time we visit a variable refrence and the variables have a constant integer value at that point we can replace the variable with the value. For dead code elimination we created a *ReverseVisitor* and removed any assignments that produce variables not used in the future. Extra care was required to handle loops and if statements. We extended our tool with a powerful heuristic of optimizing loops. We observed that loops can be reduced to simple assignments if certain conditions are respected:

- All assignments don't reference values that are modified in the loop, except themselves.
- If they reference themselves then only +,- can be used in corresponding *BinaryExpr*.
- We don't have *IfStmt* on variables in $modset_{loop}$ or *WhileStmt*
- We don't have *AssertStmt* on variables in $modset_{loop}$

Even if not all of the above are true for a loop, certain statements can still be removed. We will follow with a few examples to support our idea (LHS is the original program, RHS after optimizations are applied):

```
1. while (i < n)
     x = x + a
     i = i + c

2. x = y;
   assume(x <= 10)

   while(i < 100)
     j, b = 0
     a = 200

     while(j < 100)
       a = a - 1
       b = b + 1
       j = j + 1

     x = x + a
     y = y + b
     i = i + 1

   assert(x == y)
```

```
1. iterations = (n - i)/c
   x = x + a * iterations

2. x = y;
   assume(x <= 10)
   x = (x + 10000)
   y = (y + 10000)
   assert(x == y)
```

IX.3 Random input testing

Another approach that we tried was to run the program with random input and see if any assertion fails. This approach has the advantage of having a high chance of catching frequent bugs even on very complex programs. However, this is an under-approximation of the program and we can't guarantee that it will follow all execution paths. Because of that, rare bugs are very hard to catch. To transform the program in c++ we replaced *assume, havoc* and *assert*:

```
1. assume(expr)
2. assert(expr)
3. havoc(x)
```

```
1. if (!expr)
     return

2. if (!expr)
     printf("incorrect")
     exit(0)

3. x = random()
```

We replace the original main with a function that takes as arguments all the variables used in the original program. This wouldn't work if we had the same variable name used in different scopes so we transformed the program beforehand with a *UniqueVisitor* that assigns unique names to all variables. The new main is just an infinte loop that calls the original program with random arguments.

IX.4 Combining strategies

Because each tool has its own weaknesses, the best strategy is to run all of them in parallel and if we are lucky at least one will be certain if the program is correct or incorrect:

- BMC mode can check with certainty programs with small loops. Otherwise we need to have the technique sound so we add unwinding assertions and if the SMT solver returns *sat* because of the latter failing we return *unknown*.
- Loop Abstraction + Invariant Generation + Houdini can result in false positives if insufficient invariants are guessed so this way we can check *only* if a program is correct.
- Random input can only say if a program is incorrect and will discover that in limited cases.

When running in parallel we created an abstraction so it's easy to add a new strategy and run it in a new thread with a given time limit. In the end we collect the results from all strategies and return *CORRECT* or *INCORRECT* if at least one is certain based on the considerations above. After running on our competition test suite, by the time we are writing this, we managed to score 54/68 points.

X. Conclusion

Software verification is really hard and we generated a lot of test cases to prove that. We also tried comercially available tools like KLEE[3] and SMACK[4] and they were able to pass all the tests. However, the software written in the real world is quite different in general, not based on mathematical properties that fail for particular numbers like some of our hard test cases, so perhaps the tools were not designed for that. We were surprised how easy it is to make a program almost impossible to check, in respect to how difficult it was to prove:

```
// For any different a,b,c > 2
an = a * a * a
bn = b * b * b
cn = c * c * c
assert(cn ≠ an + bn)
```

"Certitudo durum est, et intuitionis ieiunium est."

[3] http://klee.github.io/
[4] https://github.com/smackers/smack