Software Reliability Coursework 2

`srtool`

Paul Colea *Varun Verma*

This report discusses our approach to building the Software Reliability tool for Coursework 2. We discuss each implementation task separately, outlining design decisions, challenges and key results.

SSA to SMT-LIB conversion

In the `SMTLIBQueryBuilder` class, we began by defining a function for converting a bit vector to a boolean, calling it `tobool`. Then, we used the constraints field to generate our SMT-LIB formula:

- we declare each variable present in the program as a symbolic constant using `declare-fun`;
- we generate SMT-LIB `assert` expressions for each assignment in the program;
- for clarity, we define each assert statement in the program as its own SMT-LIB; separately using `define-fun`;
- if there are any assert statements in the program, we assert the boolean OR of their negations (equivalent to the negation of their conjunction, by De Morgan's laws); if there are no assert statements in the original program, we add an `assert(false)` to make sure the SAT-solver reports the formula as an unsat (thus rightfully marking the program as correct).

We also implemented SMT-LIB translation for the rest of the supported operators in `ExprToSmtlibVisitor`, using the documentation sources provided in the spec.

SSA transformation

Our implementation of `SSAVisitor` uses a map to keep track of how many times a variable has been assigned in the program. When we visit a `Decl` node and encounter a new variable name, we add it to the map with an initial count of 0. Then, when visiting `DeclRef` nodes, we replace each variable name with its SSA form using the value stored in the map. For every `AssignStmt`, we increment the count of the LHS in the map.

Predication

We handle predication by using two `DeclRef` nodes: `globalPredicate` and `parentPredicate`, the predication of the if statement branch we are currently visiting.

`globalPredicate` will be initialized when we meet our first assume statement, and `parentPredicate` is non-null whenever we are visiting an if statement.

When we visit an `IfStmt` node, we first save the current `parentNode`. Then we create two fresh predicates, say P and Q, for the taken and not-taken branches respectively. We proceed by setting `parentPredicate` to P and visiting the taken branch, then setting `parentPredicate` to Q and visiting the not-taken branch. Finally, we set parentNode to the value it had before visiting the current node. Once we devised this scoping mechanism relying on `parentPredicate`, handling the other statements proved to be straightforward.

Two counters, `havocCounter` and `predicateCounter`, are used to produce fresh variables. `havocCounter` is incremented whenever we handle a `HavocStmt`, and `predicateCounter` is incremented when generating new predicates. We prefix each variable generated in this section with `$` in order to avoid naming conflicts with variables in the original program.

When generating the `Expr` nodes for predicated statements, we relied on the logical equivalence between $(p \rightarrow q)$ and $(!p \vee q)$. We also optimized the generated expressions in various ways. For instance, when visiting havoc(x) before any assume statements and outside of any if branch, our `PredicationVisitor` will replace it by a single assignment `x = h`, as opposed to the canonical `x = (G && P) ? h : x`.

Bounded Model Checking mode

For the BMC mode, we first extract all non-candidate invariants of the visited `WhileStmt` in a separate list, since we will have to assert these invariants whenever we unwind the loop.

We use a simple recursive method that unwinds a while statement to a given bound, called `unwindLoop`. Each iteration of this procedure returns a `BlockStmt` containing:

- assertions for all non-candidate loop invariants;
- an `IfStmt` which contains the loop's body and the result of the recursive call in its taken branch.

When reaching the base case, we interrupt further execution by inserting an assume(false), and adding unwinding assertions if performing sound BMC.

In the visitor method for while statements, we first recursively visit the body of the loop. This guarantees that before unwinding the current loop, any inner loop will have been unwound. Then we call `unwindLoop` for the current AST node, passing in the list of all non-candidate invariants and the loop's unwind bound (or the default depth, if no bound is specified). Finally, we replace the while loop in the original program with the `BlockStmt` resulting from `unwindLoop`'s computation.

Verifier mode

We implemented loop abstraction as described in the lectures, by replacing each `WhileStmt` with the following:

- assertions for each non-candidate invariant
- havoc statements for every variable in the modset
- assume statements for every variable in the modset
- an IfStmt checking the loop condition, with the following statements in its taken branch:
    - the loop body
    - assertions for each non-candidate loop invariant
    - assume(false)

Again, we need to take care to first visit the body of the current loop first, in order to handle nested loops properly.

The only information which wasn't readily available in the AST nodes was a loop's modset. We added a method in the WhileStmt node which returns a loop's modset, and used it to generate the appropriate havoc statements needed for performing loop abstraction.

Houdini

We implemented Houdini using a visitor, `HoudiniVisitor`, which extends the default visitor and only overrides the visit method for while statements.

For each candidate invariant `inv` encountered in the while statement, we do the following:

- enable it as a true invariant by performing `inv.setCandidate(false)`
- verify the program (perform loop abstraction, predication, SSA renaming, SSA to SMT translation and pass to z3) with this invariant enabled and **all other assertions in the program disabled**
- if the program is correct, we mark the candidate invariant as a true invariant and continue testing the other candidate invariants

In the version of Houdini that was presented to us, there was no mention of removing all other assertions in the code when testing whether a certain candidate loop invariant holds. However, we believe that our implementation of the algorithm achieves the goal of removing false loop invariants. By disabling all assertion statements in the program when testing each loop invariant, the only types of assertions present in the program after loop abstraction are:

- assertions belonging to true loop invariants (which cannot fail, since those expressions are known to hold). These can be either in the current loop, in loops following the current loop or in loops nested in the current loop
- the two assertions corresponding to the candidate invariant that we are currently trying to prove / disprove

If the program constructed in this way is reported as correct by z3, then we know the candidate loop invariant is a true invariant. If the program is reported as incorrect, we know that the only cause of failure are the two assertions introduced by the candidate - and thus the candidate is false.

After applying Houdini to the entire program, all the candidates which generated correct programs using the transformations above are marked as true invariants. All candidate invariants that end up generating incorrect programs remain marked as candidate - removing them from the loop's `InvariantList` would require extra computational effort.

Candidate Invariant Generation

Our candidate invariant generation code lives in the `srt.tool.invgen` package. Here we have added multiple files, which are mostly either visitors similar to previous sections, or classes with static utility methods. Consider the following fragment of Simple C:

```
int i;
int j;
i = 5;
j = 0;
k = 7;
while (i < 5 && j > 0) {
	i = i - 1;
	j = j + 1;
	k = k + 2;
}
...
```

We discuss the types of invariants generated by the heuristics we devised, with examples taken from the code snippet above:

- candidate invariants comparing each possible pair of variables in the program using different comparison operators.(`i == j, i >= j, i <= j, i != j` etc.)
- candidate invariants derived from the loop conditions only: any binary expression that compares between a variable and an expression in the loop will generate candidate invariants. In the sample code above, the loop condition has two binary expressions involving variables: `i < 5` and `j > 0`. Based on these, the `LoopConditionInvariantGenerator` will generate such candidate invariants as `i <= 5, i >= 5, j >= 0, j <= 0`, and others
- candidate invariants that compare a variable with the expression which was most recently assigned to that variable. In the example above, we note the statement `k = 7`. Our tool will generate such candidate invariants as `k >= 7, k <= 7, k == 7, k != 7` etc.
- candidate invariants derived from the loop conditions and the assignments into variables in the loop body. More specifically, the `LoopBodyAssignmentsCollector` walks the

loop body and constructs a map between each variable and assignments into that variable. The `LoopConditionCollector` (mentioned before) also provides a map between each variable and comparisons involving that variable appearing in the loop condition. In the example code, `i > 5` is a loop condition involving variable `i`, and `i = i - 1` is an assignment into variable `i` present in the loop body. The `LoopBodyAssignmentsInvariantGenerator` will generate invariants derived from these two facts, of the form: `(i > 5 - 1)`, `(i <= 5 + (-1))`, `(i != 5 + (-1))` and so on.

- candidate invariants based on ssignments into the loop body and int literals, testing modular arithmetic. For instance, based on the assignment `k = k + 2` in the code snippet, we will generate candidate invariants `k mod 2 == 0` and `k mod 2 != 0`

Although our candidate invariant generation heuristics managed to generate enough invariants to make our tests pass, we reckon that we could have optimised our tool in a number of ways, such as reducing calculations involving constants to single `IntLiteral` nodes, or removing duplicate invariants before passing them to Houdini.

All in all, our heuristics attempt to generate relatively few, meaningful invariants in order to speed up the analysis performed by Houdini. We refrained from generating many redundant candidates (e.g. by combining values with all possible binary or unary operators) and focused more on targeting typical situations arising in real programs.

Competition mode

We implemented following two optimisations to ensure that we are able to verify more demanding programs within the time limit of 60 seconds:

- Parallelisation of Houdini
- Sequential Verification of smaller sub-programs using Houdini

We use Java Executor framework with 4 threads to implement parallel version of Houdini. We exploit the fact that each invariant is independent, and does not depend on any other invariant. We can have two invariants which can be interlinked, however we can not have presence of one invariant to cause another invariant to fail. Therefore we using parallelisation for Houdini is sound.

For a program P which contains loop L, which has N generated candidate invariants. Lets assume N is 650, we create 10 copies of program P, (P_1,..,P_{10}), each one of them having 65 generated candidate invariants. We run Houdini parallelly on these 10 copies of P. Once Houdini terminates for each of these, we compute the union of the true invariants of programs (P_1,...,P_{10}). The result of the union of true invariants of P_1,...,P_{10} is the list of true invariants for the initial program P which are added back to the original program. We use batches of 65 because we tested with various different batch sizes of invariants to find the optimal batch size for the same program. Using a very high batch size for this, led to slower performance since z3

requires to test more invariants on each thread, and hence could potentially lead to not using the parallel computational power we have on a multi-core processor. On the other hand having a very small batch size leads to a larger number of Runnables initialized, which causes overheads of context-switching and hence slower performance.

The second technique we implemented involves iteratively calling parallel implementation of Houdini for a smaller program P' which only contains one loop. Since we know that z3 explores all execution paths if our program has loops to test for errors, hence testing a single loop at a time, will prune the branches exponentially which are to be explored by z3. For a program P as follows:

```
void main()
{
  int i;
  int j;

  j=0;
  i = j;
  while(i < 100)
  {
    i = i + 1;
    j=0
    while(j < 200)
    {
      j = j + 1;
    }
  }
  assert(i == j);
}
```

For the above program which contains two loops, one outer loop , and a nested loop in its body. To verify this program using Houdini, we would generate invariants for whole program P. Using this technique we first generate invariants for this sub-program:

```
void main()
{
  int i;
  int j;

  j=0;
  i = j;
  i = i + 1;
  j=0
  while(j < 200)
  {
    j = j + 1;
  }
}
```

We run parallelised Houdini on this sub-program and compute all true invariants. We eliminate the loop from the program using loop abstraction, and using the true invariants found using

Houdini. For simplicity we will assume that we only have one candidate invariant for loop above (j>=0) which is true invariant. Then we run parallelised Houdini on following program which has innermost loop abstracted:

```
void main()
{
  int i;
  int j;

  j=0;
  i = j;
  while(i < 200)
  {
    i = i + 1;
    assert((j>=0));
    havoc(j);
    assume(j>=0);
    assume(!(j>200))
  }
}
```

We remove the if statement which is generated while loop abstraction and assume the negation of the loop condition. Hence we have another sub-program with only one loop and we execute parallelised Houdini to identify all true invariants for this loop. And finally we will verify our next sub-program which will be completely without any loops and this sub-program represents the base case for this technique and we terminate. As we can see that a program can only have a limited number of loops, therefore we are guaranteed to terminate given that Houdini terminates which is true.

This technique exponentially reduces the number candidates we verify at any given time due to verifying sub-programs of initial program P. This technique also reduces the number of branches which are required to be explored by z3 in order to prove if a given formula is satisfiable or not. This is true because by having smaller sub-programs we have a smaller set of variables to generate candidate invariants and verify because adding additional 2 variables to a program increases the number of unique pairs to generate candidate invariants exponentially, and hence generating for sub programs which will have a smaller set of variables will reduce the number of candidates significantly. Therefore we observe a significant improvement in execution time to verify a given program in this mode.