Software Reliability Coursework 2
`srtool`

Deadline: Friday 7 December (midnight)

Weight: 20% of module

To be undertaken in groups of up to three

1 Introduction

For this coursework, you will build `srtool`, a tool that analyses imperative programs with the goal of either detecting, or proving the absence of, assertion violations. The tool accepts a program written in the Simple C language which has been used during the lectures,[1] and outputs a list of assertion statements that may fail.

The tool will be written in Java. You are provided with classes that parse a Simple C program into an abstract syntax tree (AST), as well as a class that interfaces with an SMT solver using the SMT-LIB (version 2) language. The Z3 solver will be used during this assignment, but other solvers will also work. You are also provided with skeleton classes implementing a basic infrastructure for `srtool`.

`srtool` should run in three modes:

1. A bounded model checking (BMC) mode where the tool finds or shows absence of assertion failures up to a certain loop unwinding depth
2. A BMC mode as in (1) which also reports an error if the unwinding depths used for the program's loops are insufficient to guarantee the absence of assertion failures
3. A static verification mode where loops are summarised using user-provided invariants, to enable proving absence of assertion failures

[1] There are some small differences between the syntax of Simple C used in the lectures and the syntax used during this assignment, these will become clear when you examine example test programs provided as part of this coursework.

2 Getting started

Download and unpack `SRTool.tgz` from CATE. In a `tcsh` shell, run the following command:

```
source /vol/lab/cs4/SoftwareReliability/coursework2.sh
```

This will put the Z3 constraint solver on your path; check that this is indeed the case by running the command `z3 -help`, which should display a list of Z3's options.

Throughout this document, we assume that the current directory is the project root, which contains the `bin`, `src`, `test`, `testloopabs`, etc. directories.

Simple C

The Simple C language used for this coursework has the grammar shown in Figure 1.

The syntax of expressions is not shown, but is a subset of the regular C expression syntax, with the unary, binary and ternary operators shown in Figure 2, as well as parenthesis. Operator precedence is the same as in C.

Notice that there are no global variables or function calls: a program is a single procedure that takes zero or more parameters and does not return anything. All parameters and local variables have `int` type, and we assume that values of type `int` are 32-bit signed integers represented in two's complement form.

Simple C does *not* feature short-circuit evaluation. For example, when evaluating a binary expression that uses the `&&` operator, both the left- and right-hand sides of the expression are always evaluated.

Variables cannot be initialised on declaration, thus the initial values of all parameters and local variables is arbitrary.

All expressions evaluate to a 32-bit signed integer. In particular *there is no Boolean type*. Operators that usually result in a boolean type (such as ==, <, etc.) yield either 0 (for *false*) or 1 (for *true*). For example, the expression `3 == 3` evaluates to 1, while the expression `3 == 2` evaluates to 0.

When interpreting an integer as a boolean, such as with the condition expression of an `if` statement, we regard the value 0 to be *false* and any other value to be *true*.

It is illegal to declare a variable with name X if a variable named X is already in scope. For example, the following code fragment is invalid:

```
int i;
{
  // Error
  int i;
}
```

```
program:
  'void' ID '(' declList? ')'
  block
;

declList: decl (',' decl)*;

block:
 '{' compoundStmt '}'
;

decl:
 'int' ID
;

compoundStmt:
 statement*
;


statement:
    block
  | 'if' '(' expr ')' statement ('else' statement)?

  | 'while' '(' expr ')'
     ('bound' '(' NUMBER ')')?
     ('inv' '(' expr ')')*
    statement

  | decl ';'
  | ID '=' expr ';'
  | 'assert' '(' expr ')' ';'
  | 'havoc' '(' ID ')' ';'
  | 'assume' '(' expr ')' ';'
  |  ';' // empty statement
;
```

Figure 1: Grammar for the Simple C language to be used during this coursework.

- Unary operators:
  `LNOT: '!';`
  `BNOT: '~';`
  `UPLUS: '+';`
  `UMINUS: '-';`
- Binary operators:
  `MULTIPLY: '*';`
  `DIVIDE: '/';`
  `MOD: '%';`
  `ADD: '+';`
  `SUBTRACT: '-';`
  `LSHIFT: '<<';`
  `RSHIFT: '>>';`
  `LT: '<';`
  `LEQ: '<=';`
  `GT: '>';`
  `GEQ: '>=';`
  `EQUAL: '==';`
  `NEQUAL: '!=';`
  `BXOR: '^';`
  `BAND: '&';`
  `BOR: '|';`
  `LAND: '&&';`
  `LOR: '||';`
- Ternary operator:
  `TERNARY: '?';`

Figure 2: Simple C operators.

You *can* declare a variable with name X once a prior variable with name X has gone out of scope. For example, the following code fragment is valid:

```
{
  int i;
}
// OK
int i;
```

A `while` statement is the same as in C, but with some extra optional features – an *unwinding bound* and a list of *loop invariants*, which are discussed below.

The `havoc` and `assume` statements behave in the manner described in the lectures.

An overview of the `srtool` infrastructure

The program entry point is `srt.tool.Main.main(String[] args)`. Most work is done in `srt.tool.SRTool.go()`.

The input Simple C file is first parsed into an AST. Each AST node has type `Node`, and all subclasses of `Node` can be found in the `srt.ast` package. The root node of an AST has type `Program`.

Each node has a `children` field that stores a `List` of child nodes. For example, the following if statement:

```
if(<C>)
 <S1>
else
 <S2>
```

is parsed into an instance of the `IfStmt` class. `IfStmt` extends `Stmt`, and `Stmt` extends `Node`. `IfStmt` has three children representing `<C>`, `<S1>` and `<S2>`. These are accessed using the accessor methods of `IfStmt`: `getCondition()`, `getThenStmt()` and `getElseStmt()`. All Nodes have accessor functions for their children and, in some cases, for other data. E.g. `IntLiteral` has a `getValue()` method.

Visitors can be used to traverse and transform an AST, so that you only need to add code for the Nodes that you are interested in. The `Visitor` interface is in the `srt.ast.visitor` package. Several implementations of `Visitor` are included in the `srt.ast.visitor.impl` package. At some point, you should study `srt.ast.visitor.impl.StupidVisitor`, as this shows many techniques that you will use. You can run `srt.test.StupidVisitorTest` as a Java application to see `StupidVisitor` transforming a Simple C program.

Visitors extend the `DefaultVisitor` class, which provides the code for traversing and transforming the AST. The `MakeBlockVisitor` class adds `BlockStmt`s (block statements) to make analysis simpler, transforming, e.g.,

```
if(1) x=1;
```

to

```
if(1) {x=1;} else {}
```

so that `IfStmt`s always have blocks as their child statements. This visitor is already used in the skeleton code in srt.tool.SRTool.

To use the `MakeBlockVisitor`, the `visit(p)` method is invoked for a `Program` p. The `MakeBlockVisitor` does not override `visit(Program p)`, so the `DefaultVisitor` implementation is executed, which visits the children of the Node recursively, starting with the first child. The order of the children of a Node can be determined by looking at the constructor of a particular `Node` subclass. Note that `DefaultVisitor` also works its way down the class inheritance hierarchy. So, when an `IfStmt` is encountered, `visit(Stmt stmt)` is called first. Since this method is not overridden by `MakeBlockVisitor`, the `DefaultVisitor` implementation then calls `visit(IfStmt ifStmt)`, with the same argument cast to the appropriate type (`IfStmt`). `MakeBlockVisitor` overrides `visit(IfStmt ifStmt)`, so it can modify the `IfStmt`.

Nodes are immutable, so "modifying" a node requires creating a new node to replace the old one. This means that to replace a single node in the AST, all predecessors must be replaced with new nodes as well, which results in a new `Program` node. To avoid having to manually recreate all predecessors, the `DefaultVisitor` can do this automatically. The `doesModify` parameter of the `DefaultVisitor` constructor must be `true` for this to work. For example, in the `MakeBlockVisitor.visit(IfStmt ifStmt)` method, a new `IfStmt` is returned to replace the `ifStmt` parameter that is being visited. The `DefaultVisitor` implementation detects that a different node was returned and replaces the parent node with a copy where the `children` field now points to the new `IfStmt`. This propagates up to the root `Program` node, which is returned by the visitor.

Constructors for the subclasses of `Node` include an optional `basedOn` parameter with type `Node`. All nodes in the original AST include token information, which provides the line and column number of the token from the source code file that the node was created from. This can be retrieved via the `getTokenInfo()` method. The `basedOn` parameter causes the new node to use the token information from the `basedOn Node` argument. The `basedOn` argument is useful when creating a new `AssertionStmt`, to ensure that errors related to this assertion are reported with respect to the line and column of the failing assertion in the original program.

Running the tool and tests

We recommend setting up an Eclipse project for this coursework, but you can work using the command line or another IDE if you prefer. You should make sure Java 1.6 or greater is used. You should add all the `.jar` files provided in `SRTool.tgz` to the classpath. You can then attach the `.zip` sources to each `.jar`, except for `jcommander.jar`, which includes the source. Z3 must also be on your path. One way to achieve this is to start Eclipse from the shell after sourcing the `coursework2.sh` file:

```
source /vol/lab/cs4/SoftwareReliability/coursework2.sh
eclipse &
```

We have also provided scripts for working from the command line. Ensure that you have sourced the `coursework2.sh` file before executing these scripts.

- `srt_build.sh` – will compile all *.java files.
- `srt_run.sh` – will run `srt.tool.Main` as a Java application. You need to provide command line arguments (use `--help` for usage.)
- `srt_stupidvisitortest.sh` – will run srt.test.StupidVisitorTest as a Java Application.
- `srt_test.sh` – will run srt.test.AllSRTTests using JUnit.
- `srt_testsmt.sh` – will run srt.test.TestSMT as a Java application.

Look in `test/1_simple/`. You will find a small set of very simple Simple C programs. For example, look at `1_bad.sc`. Recall that the assertion can fail because we assume that both function parameters and uninitialised variables can take any 32 bit value. So, if we had converted the program to SMT-LIB format and sent it to the SMT solver, `i==1` and `j==2` would be a satisfying assignment.

Open `srt.test.AllSRTTests`. This file returns all Simple C files within the `test`, `testloopabs` and `testunsound` directories (and subdirectories) so that they can be tested using JUnit. The test harness code is in `srt.test.DynTest`, which creates an instance of `srt.tool.SRTool` to run on the test file. If a test filename contains the word "good", then the result should be that no assertions can fail. If a test filename contains the word "bad", then at least one assertion should be able to fail.

Modify `AllSRTTests.java` so that we just test files in `test/1_simple`. Comment out the lines related to `testloopabs` and `testunsound` – you should re-enable these during later stages of the coursework. This allows us to just focus on the simple tests for now. Run `AllSRTTests.java` as a JUnit test. You should see that most of the tests fail. Tests with "bad" in the name currently pass because we have not yet implemented the code that adds constraints to the SMT-LIB query. Therefore, the SMT solver returns "sat" (because there are no constraints), which is what we would expect to see if an assertion could fail.

The `srt.tool.Main` class can be run as a Java Application to run the tool on a single file. In Eclipse: from the top menu choose Run -> Run configurations.... On the left, right-click "Java Application" -> New. Give the configuration a name (e.g. "srtool") and enter `srt.tool.Main` as the Main class. Click on the arguments tab. In the Program arguments text box type: `test/1_simple/2_good.sc`

Click "Apply" then "Run".

The program source will be pretty-printed to stdout followed by the result of the SMT solver – "sat". The query that was submitted to the SMT solver can be found in `queryIn.txt`. The stdout (result of the query) and stderr of the solver can be found in `queryOut.txt` and `queryErr.txt` respectively. While debugging, you may want to modify `queryIn.txt` and resubmit it to the SMT solver. In this case, you can run `srt.test.TestSMT` as a Java application in order to just submit `queryIn.txt` to the SMT solver and output the result. Note that running a JUnit test or `srt.tool.Main` will cause `queryIn.txt` to be overwritten.

3 Roadmap

Your task is to gradually extend `srtool` in a number of steps to that it can:

- handle trivial straight-line programs that are already in SSA form (Section 4)
- turn straight-line programs into SSA form (Section 5)

- apply predication to turn loop-free programs into straight-line programs (Section 6)
- apply loop unwinding necessary for the tool's BMC mode (Section 7)
- apply loop summarisation necessary for the tool's static verification mode (Section 8)

Note: You are strongly encouraged to use version control, or some backup system, throughout this exercise.

4 Turning an SSA-form program into an SMT-LIB formula

Look at `test/1_simple/2_good.sc`. This program is already in static single assignment (SSA) form. Your first task is to extend `srtool` so that it can turn a program that is already in SSA form into an SMT-LIB formula, such that the formula is satisfiable if and only if the program is not correct, as discussed in the lectures.

Study `srt.tool.SRTool` and then run the `srt.tool.Main` on `test/1_simple/2_good.sc`. The program is transformed by several visitors, after which the transformed program text is output. The visitors have not yet been implemented, so the program is unchanged. Note that you may later find it useful to print out the program after each visitor has been applied, but this is entirely up to you.

Looking in `srt.tool.SRTool`, you can see that the `CollectConstraintsVisitor` visits the program after it has been transformed. Study `srt.tool.CollectConstraintsVisitor`. This visitor collects all variable names, assignments and assert expressions. In the `visit(AssignStmt)` method, the assignments are converted to equality expressions, so:

```
x = y + 1;
```

becomes:

```
(x == y + 1)
```

The expressions are stored in two `Lists`: `transitionExprs` and `propertyExprs`. The collected variable names and expressions are then passed to an instance of `SMTLIBConverter`, which we need to (partially) implement now. Study `srt.tool.SMTLIBConverter`. The `query` field is used to store the SMT-LIB query that we will submit to the SMT solver. You should consult the lecture notes, the SMT-LIB tutorial[2] and the SMT-LIB reference[3] for information about SMT-LIB syntax. Also recall that the generated query is output to `queryIn.txt`.

[2] `http://www.grammatech.com/resources/smt/SMTLIBTutorial.pdf`

[3] `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf`

In the constructor, add suitable declarations to `query` to represent the variables in `variableNames`. The type of each variables should be a bit vector of length 32.

Next, add the transition expressions (i.e. the equalities) to the query. Recall that constraints are added using the `assert` SMT-LIB command. To convert the expressions, you should use the `ExprToSmtlibVisitor`, which returns the visited `Expr` as an SMT-LIB `String`. An instance of `ExprToSmtlibVisitor` is provided in the `exprConverter` field.

For example, say we have an assignment:

```
a = b + 1;
```

The `CollectConstraintsVisitor` will collect this and convert it to an equality expression:

```
(a == b + 1)
```

The `ExprToSmtlibVisitor` will convert this to an SMT-LIB expression (a `String`):

```
"(= a (bvadd b 1))"
```

Study `srt.tool.ExprToSmtlibVisitor`. The `BinaryExpr.ADD` operator has been implemented already. You should implement support for all the remaining operators in due course. For now, the only other binary operator you need to implement is `BinaryExpr.EQUAL`. You will also need to implement a few other methods in this class, which will become obvious when you run the tool and examine the generated `queryIn.txt`.

Finally, in the constructor of `SMTLIBConverter`, extend the query with a conjunct stating that at least one assertion can fail. You might find it useful to know that `(or X)`, where X is a single expression, is equivalent to `X`.

When you are done, there should be no errors in `queryOut.txt` and the result should be "unsat". You are advised to write some further simple tests to check that the functionality you have implemented is working as expected.

Converting to bit vectors and back

Now look at `test/1_simple/3_good.sc`. Run `srt.tool.Main` on this file. You will probably receive an error from the solver about an "invalid function application" due to a "sort mismatch". A "sort" is essentially a type. Recall that we expect all Simple C expressions to result in a bit vector of length 32. Unfortunately, SMT-LIB functions do not always result in a bit vector. In particular, equality results in a `Bool`. You will need to modify `ExprToSmtlibVisitor` so that all non-bit vector results are converted to bit vectors. Note that we have defined an SMT-LIB function `tobv32` in `SMTLIBConverter`, which you can use. You are encouraged to define more functions here so that your generated query is more readable (for debugging). You will also need

to convert certain expressions to type `Bool`, as certain functions require the arguments to be of type Bool. Finally, you will always need to convert the expressions to type `Bool` when issuing the `assert` command. Once you have done this, you should be able to run `srt.tool.Main` on `3_good.sc` and get the result "unsat" without any errors.

At this point, all of the tests in `test/1_simple` should pass. You should now implement all remaining operators in `ExprToSmtlibVisitor`. You can decode some information from the Z3 C API documentation page[4]

For example, `bvadd` is documented as the C function `Z3_mk_bvadd` (although there is little information about the types).

The official SMT-LIB documentation for these functions is here:

`http://smtlib.cs.uiowa.edu/theories/Core.smt2`

`http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2`

In these documents, the final "parameter" type refers to the result type of the function. For example,

```
(bvadd (_ BitVec m) (_ BitVec m) (_ BitVec m))
```

takes two bit vectors of length `m` and returns a bit vector of length `m`.

You should take special care when implementing the following:

- `BinaryExpr.RSHIFT` – a right-shift should preserve the sign of the number that is being shifted
- `BinaryExpr.LAND`
- `BinaryExpr.LOR`
- `UnaryExpr.LNOT`
- and all of the relational operators (e.g. $<$).

You are encouraged to write some further Simple C test cases to test your operator implementations. Recall that test filenames should contain "good" or "bad" depending on whether an assertion can fail or not. There will not be any marks for this, but it may be useful, as we will test your implementations with our own test suite and you will lose marks if operators are not correctly implemented.

Reporting which assertions failed

Note: This section can be completed at any point – you can skip it and come back to it later if you wish.

[4] `http://research.microsoft.com/en-us/um/redmond/projects/z3/group__capi.html`

Run `srt.tool.Main` on `test/asserts/asserts_bad.sc`. The tool reports that an assertion can fail, as expected, but it does not indicate which assertion. This information would be essential to a user applying a full-blown analysis tool. To solve this, you will use the `get-value` command, which was described in lectures. You will need to make the generated SMT-LIB query look similar to the following:

```
(set-logic QF_BV)
; ... omitted commands here ...
(check-sat)
(get-value (prop0 prop1 ...))
```

where the `prop`i are `Bool` variables. You must generate the `prop`i variables and extend the formula such that *prop*i is *true* if `srtool` believes that the ith assertion appearing in the program may fail. You should read the section on the `get-value` command in the SMT-LIB reference. The result of the above `get-value` command for `test/asserts/asserts_bad.sc` is:

```
((prop0 false)
 (prop1 true))
```

which indicates that the second assertion may fail. Note that the `get-value` command will return an error if the `check-sat` command returns "unsat". However, you can (and should) always include the `get-value` command for simplicity, as the error can just be ignored.

Modify `SMTLIBConverter` so that the generated query declares `prop` variables, adapts the correctness-checking formula to force a correspondence between these variables and the assertions with which they are associated, and issues `get-value` commands as illustrated above.

Implement the `getPropertiesThatFailed()` method so that a list of `Integers` are returned such that if i belongs to this list then the ith `Expr` in the `propertyExprs` list that was passed into the constructor is the guard of an assertion that may fail. The `getPropertiesThatFailed` method is called from `srt.tool.SRTool`. For each `Integer` that is returned, you should create an `AssertionFailure` object, which wraps the token information from an assertion that can fail. Note that the `CollectConstraintsVisitor` has a `propertyNodes` field: the ith `Expr` in the `properyExprs` list is the expression from the ith `AssertStmt` in the `propertyNodes` list. Thus, you can retrieve the `AssertStmt` and from the `AssertStmt` you can retrieve the token information for the assertion that can fail.

Once this is correctly implemented you should find that `srtool` reports accurate line and column information for failed assertions.

Note that when multiple assertions can fail, the SMT solver may return only a subset of the assertions that can fail. Furthermore, in the following example:

```
void f()
{
        assert(0);
```

```
        assert(0);
}
```

the tool will not consider the second assertion to be unreachable. You do not need to worry about these issues for this assignment.

5 Single static assignment

Run `srt.tool.Main` on `test/SSA/SSA_bad.sc`. The assertion should fail, yet the tool returns "unsat". Examine the transformed program and consider the constraints that are being submitted to the SMT solver. Can you see the problem? Recall that assignments become equalities.

To fix this, you will implement `srt.tool.SSAVisitor`, which should transform the program into Static Single Assignment (SSA) form as shown in the lectures. Examine `test/SSA/*.out.txt` to see how some of the test cases are transformed by the model solution. Each variable gets an index, starting at 0. This "SSA index" is incremented appropriately (e.g. at assignments). We rename a variable `var` to `var$`i, where i is the SSA index.

Implement the visit methods in `srt.tool.SSAVisitor`. Once you have done this, all tests in `test/SSA` should pass. You may wish to add further tests. Note that `if`, `havoc`, `assume` and `while` statements will not work at this stage.

6 Predication

Run `srt.tool.Main` on `test/predication/predication_bad.sc`. The assertion should fail, yet the tool returns "unsat". Examine the transformed program and consider the constraints that are being submitted to the SMT solver. The problem is that the if statement does not have any effect – it is as though the "then block" and then the "else block" (not present in this case) are both being executed, regardless of the condition.

You should now implement predication, as shown in the lectures, to support `if`, `assume` and `havoc` statements.

Implement `srt.tool.PredicationVisitor` to predicate the program. Disable (i.e., comment out in `srt.tool.SRTool`) the `SSAVisitor` while you do this, so that it is easier to examine the transformed program. You should look at `srt.ast.visitor.impl.StupidVisitor` to see some of the techniques you will need to use.

As explained in the lectures, predication involves the generation of fresh variables. You should prefix each freshly generated variable name with "$" to ensure that they cannot clash with variable names in the program. You should make sure that the conversion to

SSA form (which occurs after predication) does not cause the predicate expression to be incorrect (see "Getting predication wrong!" from Lecture 3).

Once you have implemented `PredicationVisitor` (and re-enabled `SSAVisitor`), all tests in `test/predication` should pass. `srtool` is now fully capable of handling loop-free programs!

7 Bounded model checking mode

We now consider how loops can be handled via unwinding, using BMC.

Run `srt.tool.Main` on `test/loop/loop_good.sc`. The assertion should never fail, yet the tool returns "sat". Can you see why? To solve this, we will unwind the loop a number of times, according to the given *bound* annotation.

For example, consider a loop with the following general form:

```
while(c)
{
        <body>
}
```

Unwinding this loop once yields:

```
if(c)
{
        <body>
        while(c)
        {
                <body>
        }

}
```

Unwinding the loop twice yields:

```
if(c)
{
        <body>
        if(c)
        {
                <body>
                while(c)
                {
                        <body>
                }
        }

}
```

If we wish to eliminate the loop completely, we have two choices as to how to get rid of the `while` loop that remains after unwinding. We can replace the loop with an *assumption* that the loop does not execute further. For example, if the bound is two, this leads to:

```
if(c)
{
        <body>
        if(c)
        {
                <body>
                assume(!c);
        }

}
```

In this case if the chosen bound is not sufficiently large analysis becomes *unsound*: there may be paths that would execute `<body>` additional times in the original program, but the `assume` statement inserted during unwinding will truncate these paths. This means that assertion failures that could have occurred further along such paths will be missed.

Alternatively, we can maintain soundness by adding an assertion which checks that the chosen unwinding bound was sufficient. With a bound of two, this leads to:

```
if(c)
{
        <body>
        if(c)
        {
                <body>
                assert(!c);
                assume(!c);
        }

}
```

The assertion is called an "unwinding assertion". With an unwinding assertion, if we do not choose a large enough unwinding bound the then the checker will report that the unwinding assertion can fail, even if no other assertions can fail. If we do choose a large enough unwinding bound then the assertion will be unreachable and will never be able to fail. Thus, this method ensures that our analysis is sound; if no assertions can fail, then all feasible paths through the loop were modelled and checked, and none of the assertions in the original program can fail.

A loop may specify a list of loop invariants. These should all be true every time the loop head is reached. The loop head is exactly the point at which the loop condition is evaluated. This means that the loop head is reached once even if the guard evaluates to *false* and the loop body is never executed.

Your task is to now add support for both the unsound and sound methods of loop

unwinding, as well as checking of loop invariants during loop execution.

Implement `srt.tool.LoopUnwinderVisitor` to unwind loops. If the `unwindingAssertions` field is true, then you should use an unwinding assertion followed by an assume statement, otherwise just use an assume statement. Invariants should *always* be checked, regardless of any flags. If a loop has a specified "bound" (`whileStmt.getBound() != null`) then the loop should be unwound "bound" times. Otherwise, the field `defaultUnwindBound` decides how many times to unwind a loop. Note that an unwinding bound of 0 is possible and you should consider what is needed in this case.

You can set the default unwinding depth via the `--unwind=X` command line argument, where X is the default unwinding depth to use. Unwinding assertions are enabled by default and can be disabled with `--unwinding-assertions=false`.

Note that `srt.test.AllSRTTests` automatically runs tests in `testunsound` with the `--unwinding-assertions=false` option, while tests in the `test` directory are run with `--unwinding-assertions=true`.

8 Loop summarisation

`srtool` also accepts an `--abstract-loops` option. This tells the tool to run in static verification mode, whereby loops are abstracted using *summaries*, created from invariants supplied by the user. Loop summarisation works as described in the lectures. This allows efficient verification of programs that contain loops without resorting to unwinding. This is essential for loops with large iteration counts, loops where the number of iterations depends on an unconstrained variable so that the iteration count is bounded only by the maximum signed integer value, or non-terminating loops.

Implement `srt.tool.LoopAbstractionVisitor` to abstract loops using the summarisation technique you have been taught. Note that if a loop has multiple invariants $I_1, I_2, \ldots, I_n$ this is equivalent to the loop having a single invariant, $I_1 \wedge I_2 \wedge \ldots \wedge I_n$.

Note that `srt.test.AllSRTTests` will automatically use the command line option `--abstract-loops=true` for tests in the `testloopabs` directory.

The following is an example that will fail with loop abstraction:

```
main() {
 int iterations;
 int i;
 i=0;

 // make iterations positive, but otherwise unconstrained
 assume(iterations > 0);

 while(i < iterations) {
  i = i + 1;
 }
 assert(i == iterations);
}
```

Because there are no loop invariants, the value of `i` is considered arbitrary after the loop is summarised, so the assertion fails.

Adding a loop invariant as follows constrains the value of `i` after the loop so the assertion cannot fail:

```
main() {
 int i;
 int iterations;
 i=0;

 // make iterations positive, but otherwise unconstrained
 assume(iterations > 0);

 while(i < iterations)
  inv(i <= iterations)
 {
  i = i + 1;
 }
 assert(i == iterations);
}
```

Remember that the provided loop invariant is also checked, so you cannot just make any expression a loop invariant – loop invariants must be true before entering the loop and must be maintained by the loop (must be true after every iteration).

9 Submission and marking

Make an archive, `SRToolSubmission.tgz`, containing all the source code associated with your submission (including the source code for the framework files which you should not have needed to modify). Make sure that the source code you have written is readable and elegant, and clearly but concisely commented. Upload this archive to CATE.

During marking, your submission will be auto-tested with respect to a large test suite, so make sure you have extensively tested it yourself!

Marks will be assigned as follows:

SMT formula generation	10
SSA renaming:	10
Predication:	25
Loop unwinding:	20
Loop summarisation:	20
Code style and comments	15
Total	100