WLP-based Testing of GCL Programs

Orestis Melkonian

Utrecht University, The Netherlands

o.melkonian@students.uu.nl

Abstract

The Guarded Common Language (GCL) is a programming language invented by Edsger W. Dijkstra, which, due to its simplicity, is a suitable framework for trying out different verification and testing techniques.

One instance of this is extending GCL with Hoare Logic annotations, i.e. a pre-condition and a post-condition. Using a predicate-based transformer, namely that of the weakest liberal precondition (WLP), we can try to verify whether the annotated properties actually hold.

If we choose to do bounded verification (i.e. considering program paths up to a certain length), the problem of proving the annotated behaviour becomes decidable, as our programs only include linear arithmetic and infinite array operations.

Here, we instead choose to do bounded testing, i.e. producing *meaningful* value assignments to the program's input variables and checking whether the post-condition holds. Obviously, the produced claim is not complete (as in bounded verification), since there can be an assignment to the input variables that renders the post-condition false, which we did not include in our test cases.

Last but not least, we provide an implementation of our approach, written in Haskell.

Note: The GCL presented here differs from the original one, but we will keep referring to it as GCL, for brevity's sake.

CCS Concepts **• Software and its engineering → Program verification**; **• Theory of computation → Program semantics**

Keywords Predicate transformer semantics; Weakest Liberal Pre-condition (WLP); Guarded Command Language (GCL); Hoare logic; automated testing; formal verification

1. Problem Overview

1.1 GCL syntax

Here is the basic syntax of GCL programs:

$$
\begin{array}{rcl}
\langle\text{Program}\rangle & \models & \langle\text{Name}\rangle\ (\ \langle\text{Var*}\rangle \mid \langle\text{Var*}\rangle\ )\ \langle\text{Stmt}\rangle \\
\langle\text{Stmt}\rangle & \models & \textbf{skip} \\
 & & \mid\ \textbf{assert}\ \langle\text{Expr}\rangle \\
 & & \mid\ \textbf{assume}\ \langle\text{Expr}\rangle \\
 & & \mid\ \langle\text{Var*}\rangle := \langle\text{Expr*}\rangle \\
 & & \mid\ \langle\text{Stmt}\rangle\ ;\ \langle\text{Stmt}\rangle \\
 & & \mid\ \textbf{if}\ \langle\text{Expr}\rangle\ \textbf{then}\ \langle\text{Stmt}\rangle\ \textbf{else}\ \langle\text{Stmt}\rangle \\
 & & \mid\ \textbf{while}\ \langle\text{Expr}\rangle\ \textbf{do}\ \langle\text{Stmt}\rangle \\
 & & \mid\ \textbf{var}\ \langle\text{Var*}\rangle\ \textbf{in}\ \langle\text{Stmt}\rangle \\
\langle\text{Expr}\rangle & \models & \langle\text{Literal}\rangle \\
 & & \mid\ \langle\text{Var}\rangle \\
 & & \mid\ \langle\text{Expr}\rangle\ \langle\text{BinOp}\rangle\ \langle\text{Expr}\rangle \\
 & & \mid\ (\ \textbf{forall}\ \langle\text{Var*}\rangle :: \langle\text{Expr}\rangle\ ) \\
 & & \mid\ \langle\text{Var}\rangle\ [\ \langle\text{Expr}\rangle\ ] \\
\langle\text{BinOp}\rangle & \models & + \mid - \mid \Rightarrow \mid < \mid = \\
\langle\text{Var}\rangle & \models & \langle\text{Char*}\rangle \\
\langle\text{Name}\rangle & \models & \langle\text{Char*}\rangle \\
\langle\text{Literal}\rangle & \models & \text{True} \mid \text{False} \mid 0 \ldots 9 \\
\langle\text{Char}\rangle & \models & \text{A} \ldots \text{Z} \mid \text{a} \ldots \text{z} \mid \$ \mid \_
\end{array}
$$

Using this basic syntax, we can derive useful operators, such as:

$$
\begin{gathered}
\exists x :: Px \equiv \neg(\forall x :: \neg Px) \\
e \wedge e' \equiv \neg(e \rightarrow \neg e') \\
e \vee e' \equiv (e \rightarrow e') \rightarrow e' \\
e \neq e' \equiv \neg(e = e') \\
e \leq e' \equiv \neg(e' < e) \\
e > e' \equiv e' < e \\
e \geq e' \equiv e' \leq e \\
x\ in\ [l..h] \equiv l \leq x \wedge x \leq h
\end{gathered}
$$

1.2 GCL testing

Since the syntax introduced above contains two branching statements (i.e. loops and conditionals), we will test all possible paths up to some length, which ensures the procedure terminates (the number of these paths are finite).

2. Solution

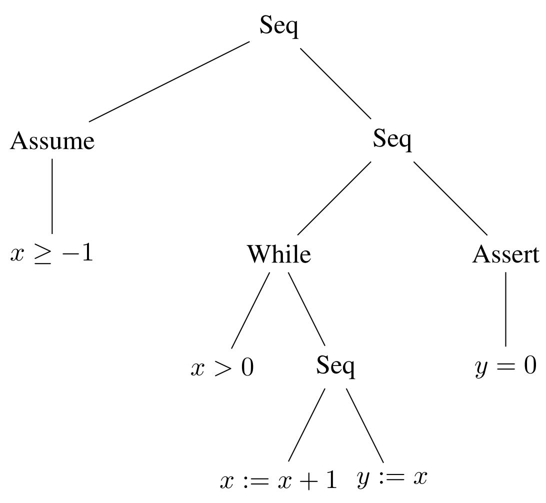
In order to demonstrate all relevant steps, we will use a simple running example, shown below:

```
{ x >= -1 }
E(x | y)
  while (x > 0) do
    x++;
    y := x
{ y = 0 }
```

2.1 Parser

The first step to testing a GCL program is to parse the actual source code, producing an internal representation that better fits our needs, namely an abstract syntax tree (AST).

For instance, the example program will be represented by this AST (partially expanded, for brevity):



2.2 Pathfinder

We then produce all possible paths (up to a given length) of the given program.

In the case of an **if-then-else** statement, we derive all paths of the *true* branch and the ones from the *false* branch.

In the case of a **while** loop, we begin unfolding it (i.e. starting from no iteration, to one iteration, etc...) and stop when we exceed the maximum length.

Returning to our running example, if we restrict our pathfinding process up to length 10, we get the following program paths:

```
-- Length: 3
assume x >= -1;
assume ~ (x > 0);
assert y = 0
-- Length: 5
assume x >= -1
assume x > 0;
x++;
y := x;
assume ~ (x > 0);
assert y = 0
-- Length: 8
assume x >= -1
assume x > 0;
x++;
y := x;
assume x > 0;
x++;
y := x;
assume ~ (x > 0);
assert y = 0
```

2.3 Renamer

To safeguard against potential name conflicts in the future, we rename all locally-scoped variables with internally generated names.

If the local variable was introduced by a **var** statement, we replace it with a fresh variable and propagate the change inside its body, as shown below:

```
  var x in                      var $0 in
    y := x       ────────→        y := $0
```

If the local variable was introduced by a **forall** expression, we replace it with a fresh variable and propagate the change in the contained expression, as shown below:

```
  assert (                      assert (
    forall x ::                   forall $1 ::
      x + 0 = x  ────────→          $1 + 0 = $1
  )                             )
```

2.4 WLP-transformer

For each one of the programs paths, we are able to calculate its WLP automatically (starting from the annotated post-condition). This is achieved by the following rules:

$$\begin{aligned}
wlp\ skip\ Q &= Q \\
wlp\ (assert\ P)\ Q &= P \wedge Q \\
wlp\ (assume\ P)\ Q &= P \Rightarrow Q \\
wlp\ (x := e)\ Q &= Q[e/x] \\
wlp\ (S\ ;\ S')\ Q &= wlp\ S\ (wlp\ S'\ Q) \\
wlp\ (var\ x\ in\ S)\ Q &= (forall\ x :: wlp\ S\ Q)
\end{aligned}$$

For instance, the WLP of our example program on the program path of length 5 would be:

$$x \geq 1 \Rightarrow x > 0 \Rightarrow \neg(x + 1 > 0) \Rightarrow x + 1 > 0$$

2.5 Normalizer

The generalization of the structure of the resulting WLP will always have the form:

$$A_1 \Rightarrow A_2 \Rightarrow \cdots \Rightarrow A_n \Rightarrow G$$

The above is equivalent to:

$$A_1 \land A_2 \land \cdots \land A_n \Rightarrow G$$

We can normalize this to a list of assumptions and a single final goal as such:

$$\begin{array}{c} A_1 \\ A_2 \\ \vdots \\ A_n \\ \hline G \end{array}$$

Again a concrete instance of our running example:

$$\begin{array}{c} x \geq 1 \\ x > 0 \\ \neg(x+1>0) \\ \hline x+1>0 \end{array}$$

2.6 TestGenerator

At this point, we could start randomly generating inputs and checking whether the logical claim holds. This would be highly inefficient though, as a proportion of these random inputs will falsify at least one of the assumptions, leading to the overall logical formula being valid.

Hence, we only care about *relevant* inputs, meaning that they render the assumptions valid. In other words, the test cases we wish to generate are the models that satisfy the left hand side of the implication, i.e.

$$A_1 \land A_2 \land \cdots \land A_n$$

At this point, it is a good idea to employ the power of a back-end SMT solver (e.g. Z3/MathSAT) to retrieve these models.

In the particular case of our example program path of length 5, the solver would not be able to find a model that satisfies $x \geq 1 \land x > 0 \land \neg(x+1>0)$, since it is not satisfiable. Thus, this program path is *infeasible*, meaning there are no inputs we can feed the program, such that the control flow leads to this path. In such cases, we can simply ignore the path altogether and continue testing other paths.

2.7 GoalProver

Assuming the path we are testing is *feasible* and an SMT solver has provided us with a *relevant* test case, we can now assign the model to our goal, i.e. substitute all (free) variables with the model's value assignments.

Unfortunately, it is not trivial to prove whether the resulting formula holds, due to the existence of quantifiers. Therefore, we again rely on the existing SMT solver to prove the goal. The caveat here is that the chosen Haskell library that interfaces with well-known SMT solvers, requires the formulas to be in *prenex normal form*, i.e. a proposition where all quantifiers appear in the beginning of the formula.

Since our logical connectives consist only of negation and implication, we apply these simple rewriting rules to convert our goal to *prenex normal form*:

$$\begin{aligned} \neg\exists x :: \phi &\equiv \forall x :: \neg\phi \\ \neg\forall x :: \phi &\equiv \exists x :: \neg\phi \\ (\forall x :: \phi) \Rightarrow \psi &\equiv \exists x :: \phi \Rightarrow \psi \\ (\exists x :: \phi) \Rightarrow \psi &\equiv \forall x :: \phi \Rightarrow \psi \\ \phi \Rightarrow (\exists x :: \psi) &\equiv \exists x :: \phi \Rightarrow \psi \\ \phi \Rightarrow (\forall x :: \psi) &\equiv \forall x :: \phi \Rightarrow \psi \end{aligned}$$

We can now conclude our testing process, by aggregating over the results of all examined program paths and giving an overall answer.

3. Handling Arrays

So far we have eluded the way in which we handle (infinite) arrays, since our running example did not contain any array accesses.

Reasoning about arrays is possible by treating them as uninterpreted functions, i.e. an access of array α at index ι is equivalent to $\alpha(\iota)$, where α is an uninterpreted function and, thus, the only axiom we have is that it is deterministic, i.e.:

$$\forall x, y :: x = y \Rightarrow \alpha(x) = \alpha(y)$$

An important thing to note here is that, as the symbols that represent the arrays are uninterpreted, the models we retrieve from the assumptions will not contain a value assignment to them. Nonetheless, the assumptions involving them will still contribute to finding *relevant* inputs. Furthermore, we need to propagate assumptions involving arrays to the goal, essentially reversing the normalizer's step.

To demonstrate reasoning about arrays, let us see an example program which finds the index of the minimum element of an array within a specified range:

```
{ i < N }
MININD(a, i, N | r)
  var min, i0 in
    i0 := i;
    min, r := a[i0], i0;
    while i0 < N do
      if a[i0] < min then
        min, r := a[i0], i0
      else
        skip
      fi;
      i0++
{ (forall j :: j in [i..N) ==> a[r] <= a[j]) }
```

Let's examine the program path with two iterations of the **while** loop statement, where in the first one we take the *false* branch of the **if-then-else** statement and, in the second time, we take the *true* branch:

```
{ i < N }
var min, i0 in
  i0 := i;
  min, r := a[i0], i0;
  assume i0 < N;
  assume ~ (a[i0] < min);
  skip;
  i0++;
  assume i0 < N;
  assume a[i0] < min;
  min, r := a[i0], i;
  i0++
  assume ~ (i0 < N)
{ (forall j :: j in [i..N) ==> a[r] <= a[j]) }
```

The resulting (normalized) WLP will be:

$$\frac{\begin{array}{c} i < N \\ i < N \\ \neg(a[i] < a[i]) \\ i+1 < N \\ a[i+1] < a[i] \\ \neg(i+2 < N) \end{array}}{\forall j \; in \; [i..N) \Rightarrow a[i+1] \leq a[j]}$$

A possible model that validates the assumptions would be $i = 0$ and $N = 2$ (the uninterpreted symbol a will not be assigned a value).

We now merge the goal with the assumptions which reason about arrays, leaving us with the following goal to prove:

$$\neg(a[0] < a[0]) \Rightarrow a[1] < a[0] \Rightarrow \forall j \; in \; [0..2) \Rightarrow a[1] \leq a[j]$$

We can easily see that the proposition above is valid, but would be impossible to prove if we hadn't carried the selected assumptions into our goal.

4. Extensions

In this section, we will discuss extensions we made to the basic functionality discussed so far, in order to increase the expressiveness of our language.

4.1 Array Assignment

If we allow assigning values to specific indices of an array, we must first extend the syntax with array assignment statements, conditional and **repby** expressions, as follows:

$$\begin{array}{lll} \langle\text{Stmt}\rangle & \models & \ldots \\ & & \mid \; \langle\text{Var}\rangle \, [\langle\text{Expr}\rangle] := \langle\text{Expr*}\rangle \\ \langle\text{Expr}\rangle & \models & \ldots \\ & & \mid \; (\langle\text{Expr}\rangle \rightarrow \langle\text{Expr}\rangle \mid \langle\text{Expr}\rangle) \\ & & \mid \; \langle\text{Var}\rangle[\langle\text{Expr}\rangle \rightarrow \langle\text{Expr}\rangle] \end{array}$$

We can now replace assignments to specific array indices with a copy of the whole array, where the element at some index has been changed to the assigned value, as follows:

$$(a[i] := e) \equiv (a := a[i \rightarrow e])$$

While calculating the WLP of such statements, we will need to replace occurrences of arrays with conditional expressions regarding their indices, based on previous assignments (e.g. $a[i] := e$):

$$P(a[x]) \equiv P(x = i \rightarrow e \mid a[x])$$

As mainstream variants of first-order logic do not support the introduced conditional expressions, we preprocess the logical formulas before giving them to the SMT solver. Assuming a conditional expression $(x = i \rightarrow e \mid e')$, we replace it with a freshly generated free variable y and add two assumptions, namely:

$$x = i \Rightarrow y = e \qquad (1)$$

$$x \neq i \Rightarrow y = e' \qquad (2)$$

As an example, consider a program that swaps positions on two elements of an array:

```
{ true }
SWAP(a, i, j | r)
  r := a;
  r[i] := a[j];
  r[j] := a[i]
{ a[i] = r[j] /\
  a[j] = r[i] /\
  (forall k :: k != i /\ k != j ==> a[k] = r[k])
}
```

We only need to consider one program path, since there are no branching statements in the original program. The resulting goal will be:

$$\begin{array}{l} a[i] = (j = j \rightarrow a[i] \mid (j = i \rightarrow a[j] \mid a[j])) \wedge \\ a[j] = (i = j \rightarrow a[i] \mid (i = i \rightarrow a[j] \mid a[i])) \wedge \\ \forall k :: k \neq i \wedge k \neq j \Rightarrow \\ \quad a[k] = (k = j \rightarrow a[i] \mid (k = i \rightarrow a[j] \mid a[k])) \end{array}$$

4.2 Loop Invariants

Another extension we have made is to allow the programmer to annotate his/her loops with invariants, which are then asserted through each iteration, alongside the final goal. This basically allows for finer-grained verification.

Specifically, we insert an assertion before the loop statement, making sure the invariant will hold after the initial part of the program has been executed. Moreover, we insert an assertion at the end of every iteration, making sure the invariant holds in each unfolding of the loop.

Firstly, we need to extend the syntax of while statements, as such:

$$\begin{array}{lll} \langle\text{Stmt}\rangle & \models & \ldots \\ & | & \{\langle\text{Expr}\rangle\}\textbf{while } \langle\text{Expr}\rangle \textbf{ do } \langle\text{Stmt}\rangle \end{array}$$

Let us now consider a program, which has an annotated loop:

```
{ x > 2 }
LOOP_INVARIANT(x|y)
  var k in
    k := 2;
    { x >= 0 }
    while k > 0 do
      k--; x--
{ x >= 0 }
```

The only *feasible* path is the one in which we have two loop iterations, amongst which we have inserted assertions proving the invariant:

```
assume x > 2;
var k in
  k := 2;
  assert x >= 0;
  assume k > 0;
  k--; x--;
  assert x >= 0;
  assume k > 0;
  k--; x--;
  assert x >= 0;
  assume ~ (k > 0)
assert x >= 0
```

4.3 Program Calls

Another useful feature we added is the ability to combine programs, namely non-recursively calling a program inside another program. First, we need the extend the syntax of GCL assignments to accommodate this feature, as such:

$$\begin{array}{lll} \langle\text{Stmt}\rangle & \models & \ldots \\ & | & \langle\text{Var*}\rangle := \langle\text{Name}\rangle\ (\langle\text{Expr*}\rangle\ ) \end{array}$$

We simply do *black-box testing* and replace such assignments, as follows:

```
-- Black-box program
{P} Prog(i, j | k, l) {Q}
-- Current program
x, y := Prog(a, b)
-- Final transformation
var a', b', x', y' in
  a', b' := a, b;
  assert P[a,b/i,j];
  assume Q[x',y'/k,l];
  x, y := x', y'
```

Note: Since no program-call recursion is allowed, a program is only allowed to call programs declared above it in the source file.

4.4 Multiple Relevant Test Cases

Up to this point, we have only examined a single test case, namely one model that satisfies the assumptions. This is quite problematic, as can be seen in the example below:

```
{ x >= 0 }
RELEVANT(x|y)
    y := x + 1
{ y < 2 }
```

If we only examine a single model, we would (most likely) get the model $x \mapsto 0$, which passes the test, whilst the model $x \mapsto 1$ does not!

Therefore, instead of retrieving just one model from the SMT solver, we retrieve many of them, with each subsequent one excluding the value assignments of the previous. In the example program above, we would first have this (normalized) goal:

$$\frac{x \geq 0}{x + 1 < 2}$$

Assuming the model returned to us is $x \mapsto 0$ (which makes the goal valid), we proceed to the next test case with the following goal:

$$\frac{\begin{array}{c} x \geq 0 \\ x \neq 0 \end{array}}{x + 1 < 2}$$

Assuming we now get the model $x \mapsto 1$, we can prove that the goal is invalid, hence finding a bug in the program with just one additional test case.

5. Conclusion

Summing up, we have presented an efficient way of testing properties of programs written in an expressive variant of GCL, involving loops, conditionals, local variables, linear arithmetic, arrays and program calls.

This properties are expressed in Hoare logic, namely a pre-condition which is assumed to hold initially, a post-condition which should hold at the end of the program's execution and several invariants annotating the program's loop statements.

We conceptuality our testing approach with a software tool written in Haskell, using the *Data.SBV* library to interface with a mainstream SMT solver. As the testing we conduct here is bounded, the software tool allows you to parametrize the maximum path length that should be investigated. Additionally, it is possible to control the desired number of test cases generated for each path, enabling the user to trade off computation time with a more credible test result.