Program Verification 2012–2013
Project 3: Bytecode Verification Engine

Danny Bergsma Jaap van der Plas

April 4, 2013

1 Introduction

For PV Project 3, we have designed and implemented a weakest-precondition calculus for a bytecode language. [3] We use this calculus to verify specifications (pre- and post-condition combinations) for programs in this language: the specification is correct if and only if the precondition implies the weakest precondition (WP) with respect to the given postcondition and program (we call this implication the verification condition - VC); the WP is calculated by subsequently applying the rules from the defined WP calculus (see Section 2.3). We check the validity of the VC by converting it to Data.SBV [1] format and subsequently calling SBV's prove function; SBV interfaces with Microsoft's theorem prover (SMT solver) Z3 [2]. Z3/SBV returns whether the VC is valid or not; if the VC is invalid, Z3/SBV gives a counterexample.

We have implemented the verifier in Haskell, using attribute grammars [4]. This report will explain our solution with emphasis on the conceptual ideas. For (more) implementation details, we refer to the Haddock (HTML) documentation and the (documented) code. The included README elaborates on practical matters, like building and running the verifier, how to test the included examples, et cetera. The README also lists all implemented features.

Section 2 explains our solution for the base task. Our 'return from anywhere' extension is described in Section 3; our bounded verification extension, to deal with loops, is described in Section 4.

2 Base task

This section presents our solution for the base task: designing and implementing a verification engine for a bytecode language.

For a more detailed description of the base task, we refer to the assignment [3].

2.1 Programs

The assignment [3] already contained a description of the syntax and semantics of the bytecode language. The representation in Haskell is straightforward; the file `PV3/Program/ProgramAST.ag` contains the defined AST.

We decided that the type of parameters, local variables, literals and return values is either *bool* or *int*. To compare two *bool*s, we added the new instruction EQUIV; we cannot use the existing EQ instruction, as our substitution implementation (Section 2.3) needs to know the type of the terms $stack_T$ and $stack_{T-1}$ that are substituted for the term $stack_T$ in the WP rule for EQ/EQUIV (see Section 2.3). If we compare two *bool*s, both terms would be of type *bool*; if we compare two *int*s, both terms would be of type *int*. The required type information is not available from a type checker, as we do not have one, so we decided to incorporate the type information explicitly into the syntax.

We have not developed a parser for programs.

2.2 Conditions

The assignment [3] already outlined what conditions should be accepted by the verifier: simple expressions and first-order universal and existential quantifications. We represent conditions (boolean expressions) again as ASTs, defined in the file `PV3/Condition/ConditionAST.ag`. More complex boolean expressions can be built out of simpler ones, using the most common logical connectives ($\vee$, $\wedge$, ...). Relational operators like $>$, $\geqslant$, ... are used to build boolean expressions out of integer expressions; more complex integer expressions can be built out of simpler ones, using addition, subtraction, multiplication and (integer) division. Conditions can also consist of integer and boolean literals. Return values (*return*) and pre-invocation parameter values (a_i) are explicitly encoded by a separate data constructor.

(Intermediate) Weakest preconditions are represented by the same AST type. As (intermediate) WPs may reference to internal (stack) data, like $stack_T$, we defined the attribute grammar `PV3/Condition/External.ag` that checks whether a given condition is "external", i.e. it does *not* refer to internal (stack) data; this is useful for verification purposes, see Section 2.4.

2.2.1 Quantifications

Universal and existential quantification can be used in conditions, but we doubt the usefulness. We included the example `PV3/Examples/Quantification.hs`. Its program's higher-level counterpart is:

```
int p (int a0) {
  return a0;
}
```

The specification for the example is:

$$pre : 6 \leqslant a_0 \leqslant 10$$
$$post : \exists\, 6 \leqslant x \leqslant 10 : return \equiv x$$

When we verify this example, we indeed prove the specification correct: `Q.E.D.` When we change the postcondition to

$$post : \exists\, 7 \leqslant x \leqslant 10 : return \equiv x$$

we get an counterexample, indicating that the specification is incorrect:

```
Falsifiable. Counter-example:
  a0 = 6 :: SInteger
```

2.2.2 Conversion to SBV

Converting from conditions to SBV format is done by the attribute grammar `PV3/Condition/ConvertToSBV.ag`. This AG builds a `Symbolic` computation. Essentially, every operator is replaced with its symbolic counterpart. Variables, including (pre-invocation) parameter values, are looked up in an environment (`Map`), to ensure that each occurrence of a variable reference refer to the same actual variable. The environment is pre-filled with all parameters ($a_0, \ldots$). An (universal/existential) quantification inserts a new variable into this environment and passes the updated environment down to the (boolean) subexpression.

Our decision to accept boolean and *int* parameters complicates this pre-filling of the environment, as type information is not available. Our first strategy was to insert each parameter twice, once into the boolean environment and once into the *int* environment (we need, at least conceptually, two environments, as our condition is typed). Unfortunately, when the verifier gave an counterexample, it listed all parameters twice, as there are two "versions" (a boolean and *int* one). It also listed parameters which did not occur at all in the VC, as the pre-filling was done based on the number of variables *specified by the program*; as these "non-occurring" parameters are irrelevant for a(n eventual) counterexample, we would like to omit them.

Our solution to these issues is to extract type information from the verification condition. This is done by the attribute grammar `PV3/Condition/Extract.ag`. This AG returns a tuple: the first element a set of indices of all parameters that are used as boolean, the second element a set of indices of all parameters that are used as *int*s. If the intersection of these two sets is non-empty, one or more parameters are used as boolean *and* integer, and an error is issued. In the other case, we now know exactly how to fill the boolean and *int* environments correctly.

We do not insert the return value into either of the environments, as a valid VC does not refer to the return value; if it does, the precondition refers to the return value, which makes no sense, or the postcondition refers to the value, but the program does not always (possibly never) return a value at all. This is at least inconsistent; programs in our bytecode language that do not return a value are also pretty useless, as there is no global state the program could alter. If the VC does refer to the return value, an error is given, as it cannot be found in either of the environments; the same error is issued when any other non-existing variable is being referred.

We have not developed a parser for conditions.

2.3 Weakest precondition rules

Essentially, our weakest precondition rules rewind the effects of instructions. Some of the rules were already given in the assignment [3].

$$\texttt{SETLOCAL}\ k\ x =_{sem} \{loc_k := x\}$$
$$wp\ (\texttt{SETLOCAL}\ k\ x)\ Q\ =_{def}\ Q[x/loc_k]$$

$$\texttt{LOADLOCAL}\ k =_{sem} \{T := T+1\ ;\ stack_T := loc_k\}$$
$$wp\ (\texttt{LOADLOCAL}\ k)\ Q\ =_{def}\ (Q[loc_k/stack_T])[(T+1)/T]$$

$$\texttt{STORELOCAL}\ k =_{sem} \{loc_k := stack_T\ ;\ T := T-1\}$$
$$wp\ (\texttt{STORELOCAL}\ k)\ Q\ =_{def}\ (Q[(T-1)/T])[stack_T/loc_k] \wedge T \geq 0$$

$$\texttt{LOADPARAM}\ k =_{sem} \{T := T+1\ ;\ stack_T := param_k\}$$
$$wp\ (\texttt{LOADPARAM}\ k)\ Q\ =_{def}\ (Q[param_k/stack_T])[(T+1)/T]$$

$$\texttt{STOREPARAM}\ k =_{sem} \{param_k := stack_T\ ;\ T := T-1\}$$
$$wp\ (\texttt{STOREPARAM}\ k)\ Q\ =_{def}\ (Q[(T-1)/T])[stack_T/param_k] \wedge T \geq 0$$

$$\texttt{PUSHLITERAL}\ l =_{sem} \{T := T+1\ ;\ stack_T := l\}$$
$$wp\ (\texttt{PUSHLITERAL}\ l)\ Q\ =_{def}\ (Q[l/stack_T])[(T+1)/T]$$

$$\texttt{POP} =_{sem} \{T := T-1\}$$
$$wp\ (\texttt{POP})\ Q\ =_{def}\ Q[T-1/T] \wedge T \geq 0$$

$$\texttt{ADD} =_{sem} \{stack_{T-1} := stack_{T-1} + stack_T\ ;\ T := T-1\}$$
$$wp\ (\texttt{ADD})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} + stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{MIN} =_{sem} \{stack_{T-1} := stack_{T-1} - stack_T\ ;\ T := T-1\}$$
$$wp\ (\texttt{MIN})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} - stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{MUL} =_{sem} \{stack_{T-1} := stack_{T-1} * stack_T\ ;\ T := T-1\}$$
$$wp\ (\texttt{MUL})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} * stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{EQ} =_{sem} \{stack_{T-1} := stack_{T-1} \equiv stack_T \ ; \ T := T-1\}$$
$$wp\ (\texttt{EQ})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} \equiv stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{LT} =_{sem} \{stack_{T-1} := stack_{T-1} < stack_T \ ; \ T := T-1\}$$
$$wp\ (\texttt{LT})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} < stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{LTE} =_{sem} \{stack_{T-1} := stack_{T-1} \leq stack_T \ ; \ T := T-1\}$$
$$wp\ (\texttt{LTE})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} \leq stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{GT} =_{sem} \{stack_{T-1} := stack_{T-1} > stack_T \ ; \ T := T-1\}$$
$$wp\ (\texttt{GT})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} > stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{GTE} =_{sem} \{stack_{T-1} := stack_{T-1} \geq stack_T \ ; \ T := T-1\}$$
$$wp\ (\texttt{GTE})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} \geq stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{EQUIV} =_{sem} \{stack_{T-1} := stack_{T-1} \leftrightarrow stack_T \ ; \ T := T-1\}$$
$$wp\ (\texttt{EQUIV})\ Q\ =_{def}\ (Q[(T-1)/T])[(stack_{T-1} \leftrightarrow stack_T)/stack_{T-1}] \wedge T \geq 1$$

$$\texttt{return} =_{sem} \{return := stack_T \ ; \ T := T-1\}$$
$$wp\ (\texttt{return})\ Q\ =_{def}\ (Q[T-1/T])[stack_T/return] \wedge T \geq 0$$

$$wp\ (s_1;\ s_2;\ \ldots;\ s_n)\ Q\ =_{def}\ wp\ s_1\ (wp\ s_2\ (\ldots\ (wp\ s_n\ Q)))$$

$$wp\ (\texttt{iftrue}\ s_1\ \texttt{else}\ s_2)\ Q\ =_{def}\ \begin{array}{l}((wp\ s_1\ Q)[T-1/T] \wedge stack_T) \vee \\ ((wp\ s_2\ Q)[T-1/T] \wedge \neg stack_T)) \wedge T \geq 0\end{array}$$

$$wp\ (\texttt{prog P}\ n\ \texttt{s})\ Q\ =_{def}\ ((wp\ s\ (Q \wedge T \equiv -1))[a_0/param_0][a_1/param_1]\ldots[a_{n-1}/param_{n-1}])[-1/T]$$

Note that we enforce that the stack is empty after returning (which includes a `POP`) by incorporating the conjunct $T \equiv -1$ into the WP rule for programs; we enforce that the stack is empty before entering the program by substituting -1 for T in the same rule. The weakest precondition of the body of the program may reference to (current) parameter values ($param_i$). As these values are now equal to the pre-invocation ones, we substitute a_i for each $param_i$; they have now the same name as pre-invocation parameter values, if present, in the given pre- and postcondition.

The WP rules are straightforwardly implemented in the attribute grammar `PV3/WP.ag`. The actual substitutions are done by the attribute grammar `PV3/Condition/Replace.ag`; this AG is called from within the AG `WP.ag`.

2.4 Verification

The actual verification consists of these steps:

1. `WP.ag`: calculating the weakest precondition with respect to the given postcondition and program
2. `PV3/Verification.hs`: generating the VC: precondition → wp from step 1
3. `External.ag`: checking the VC for being "external" (see below)
4. `Extract.ag`: extracting the parameter type information from the VC
5. `Verification.hs`: pre-filling the environment(s)
6. `ConvertToSBV.ag`: converting the VC to a `Symbolic` SBV computation, using the pre-filled environment(s)
7. `Main.hs`: calling SBV's prove function to prove the VC from step 6 correct
8. SBV: calling Z3 and returning the result of the verification (correct or not); if not correct, giving counterexample
9. `Main.hs`: outputting result of the verification and counterexample, if present

The function

verify :: *Cond* -- Precondition of specification.
→ *Program* -- Program of specification.
→ *Cond* -- Postcondition of specification.
→ *Symbolic SBool* -- Resulting verification condition.

from `Verification.hs` drives the first part (the first six steps) of the verification; $verify$ is called by the *main* function in `Main.hs`.

Consider the malformed program:

```
P (0) {
  return ;
}
```

with as postcondition:

$$post : return \equiv 2$$

The weakest precondition with respect to this program and postcondition refers to $stack_T$, as the *return* term in the postcondition is replaced with $stack_T$. This indicates that the program is malformed and no (non-trivial) specification for the program can be proven correct. Every weakest precondition (and, consequently, VC), with respect to a given program, that refers to internal (stack) data indicates that the given program is malformed and that no (non-trivial) specification for this program can be proven correct. So, if the VC is not "external", we do not need to convert it to SBV and give it to Z3; instead, we issue the error that the given program is malformed.

Note that a counterexample is only given when one or more (pre-invocation) values (a_i) and/or universally/existentially quantified variables appear in the (invalid) VC; if this is not the case, the invalidity of the VC does not depend on these values/variables and `Falsifiable` is the only output given.

2.5 Examples

We have included some examples that illustrate the workings of our verifier. Those can be found in the `PV3/Examples` directory.

2.5.1 `Simple`

This is a very simple example. The program just returns 1:

```
int p () {
  return 1;
}
```

The specification is:

$$pre : true$$
$$post : return \equiv 1$$

When we verify this example, the specification is indeed proven correct: `Q.E.D.` When we change the postcondition to

$$post : return \equiv 0$$

the output of the verifier becomes `Falsifiable`. As no pre-invocation values/variables appear in the VC, no counterexample is given.

2.5.2 `Assignment`

This is the example from the assignment:

```
int p (int a0, int a1) {
  var x0 = 10;
  if (a1+a2 == x0)
    return 1;
  else
    return -1;
}
```

The specification is:

$$pre : true$$
$$post : (a_0 + a_1 \equiv 10) \leftrightarrow (return \equiv 1)$$

When we verify this example, the specification is indeed proven correct: `Q.E.D.` When we change the postcondition to

$$post : (a_0 + a_1 \equiv 11) \leftrightarrow (return \equiv 1)$$

the output of the verifier becomes:

```
Falsifiable. Counter-example:
  a0 = 10 :: SInteger
  a1 = 0 :: SInteger
```

Now pre-invocation values do appear in the VC: a counterexample is given.

When we we change the specification to

$$pre : a_0 + a_1 \equiv 10$$
$$post : return \equiv 1$$

the updated specification is also proven correct. But if you change the precondition back to the original one (*true*), we get a counterexample:

```
Falsifiable. Counter-example:
  a0 = 11 :: SInteger
  a1 = 0 :: SInteger
```

3 'Return from anywhere' extension

In the base task, a return instruction could only appear as the last instruction in a program. In this extension we will remove this restriction.

3.1 Updated WP rules

The extension is quite simple: We give to each statement/instruction not only a(n intermediate) WP, but also the program body's postcondition input "given" by the program WP rule; this postcondition is identical to the postcondition from the specification, apart from the additional $T \equiv -1$ conjunct (see Section 2.3). The return instruction always uses the body's postcondition; the other instructions use the "normal" (intermediate) WP, as was the case in the base task.

The input to the WP rules is now a tuple: the first element (Q) the "normal" (intermediate) WP, the second element the body's postcondition. The output is now also a tuple: the first element the "new" (intermediate) WP, the second element is used to pass the body's postcondition to the "next" (previous) statement.

The return instruction WP rule is updated to:

$$\texttt{return} =_{sem} \{return := stack_T \; ; \; T := T - 1\}$$
$$wp\ (\texttt{return})\ (Q, Q') \ =_{def} \ ((Q'[T-1/T])[stack_T/return] \wedge T \geq 0, Q')$$

The rules for the other instructions remain the same, except it copies the body's postcondition Q':

$$\texttt{SETLOCAL}\ \ k\ \ x =_{sem} \{loc_k := x\}$$
$$wp\ (\texttt{SETLOCAL}\ \ k\ \ x)\ (Q, Q') \ =_{def} \ (Q[x/loc_k], Q')$$

3.2 Examples

3.2.1 `ReturnFromAnywhere`

This is the example from the assignment, slightly altered:

```
int p (int a0, int a1) {
  var x0 = 10;
  if (a0+a1 == x0)
    return 1;
  return -1;
  return 1;
}
```

The specification is:

$$pre : true$$
$$post : (a_0 + a_1 \equiv 10) \leftrightarrow (return \equiv 1)$$

Note that the last `return` statement is dead code.

When we verify this example, the specification is indeed proven correct: `Q.E.D.` When we change the postcondition to

$$post : (a_0 + a_1 \equiv 11) \leftrightarrow (return \equiv 1)$$

the output of the verifier becomes:

```
Falsifiable. Counter-example:
  a0 = 10 :: SInteger
  a1 = 0 :: SInteger
```

3.2.2 `AnotherExample`

This example uses almost all instructions from the bytecode language. It also uses the 'return from anywhere' extension:

```
int p (int a0, int a1) {
  var x0 = 10;
  var x1 = 10;
  if (10 == 10)
    x0 = 10;
  else
    x0 = 11;
  a1 = a0 + 10;
  if (a0 >= 10) {}
  else
    return 0;
  return a1 * 2;
}
```

The specification is:

$$pre : true$$
$$post : (a_0 \geqslant 10 \rightarrow return \equiv (a_0 + 10) \times 2) \wedge (a_0 < 10 \rightarrow return \equiv 0)$$

When we verify this example, the specification is indeed proven correct: `Q.E.D.` When we change the postcondition to

$$post : (a_0 \geqslant 10 \rightarrow return \equiv (a_0 + 10) \times 3) \wedge (a_0 < 10 \rightarrow return \equiv 0)$$

the output of the verifier becomes:

```
Falsifiable. Counter-example:
  a0 = 10 :: SInteger
```

4 Bounded verification extension

To verify programs with loops, you usually have to provide a loop invariant. This is not needed when you do bounded verification: you only verify executions whose number of instructions $\leqslant$ the (upper) bound.

4.1 Updated WP calculation

We added a while-statement to `ProgramAST.ag` and added the while WP rules from the assignment [3] to `WP.ag`. To verify all executions whose number of instructions does not exceed the bound, we determine all paths whose number of instructions does not exceed this bound. The path (and, consequently, number of instructions) depends on the number of iterations of each loop. So, we determine all "loop iteration configurations" that do not exceed the upper bound.

To this end, we first calculates the number of loops, including inner ones, in our program. This is done by the attribute grammar `PV3/Program/NumberOfLoops.ag`. We then pass "configurations" to the `WP` AG, which returns the WP corresponding to this "configuration". The AG also returns the total number of instructions, which can then be compared to the bound. The implicit `POP`s in `if` and `while` statements are counted for one instruction.

We will demonstrate the generation of loop configurations with an example. Consider example `Bounded4`, which has two outer loops (L1 and L2) and one inner loop (L1-L1), and a bound of 30 instructions. The first configuration (L1, L1-L1, L2) we try, is (0, 0, 0): no loop is entered and the number of instructions is 16. This is smaller than the bound, so we need to consider this path. Then, we increase the last element in our configuration: (0, 0, 1); loop L2 is now entered exactly once. The `WP` AG returns 28 instructions, which is still smaller than 30, so we need to consider this path as well. To this end, we `AND` both WPs. (Note that this is not needed here, as the last WP "incorporates" the first WP. As this optimisation would make our implementation more complex, we decided to skip it.) We increase the last element again: (0, 0, 2). The corresponding number of instructions, 40, now exceeds the bound, so we will not consider this path. We also stop with increasing the last element, as the number of instructions would only become more larger than the bound.

We now return to the original configuration (0, 0, 0) and increase the *second* element: (0, 1, 0). The corresponding number of instructions, 16, is the same as the number of instructions corresponding to the configuration (0, 0, 0); as the outer loop L1 of the inner loop L1-L1 is never entered, the inner one is not "enabled", so it makes no sense to increase L1-L1's iterations. Our generation algorithm remembers the "previous" length; if the "new" length is equal to the "previous" one, it detects a non-enabled inner loop and stops increasing the number of iterations for this inner loop.

Consequently, we return to the original configuration (0, 0, 0) and increase the *first* element: (1, 0, 0). The corresponding number of instructions, 28, is now *not* the same as the number of instructions corresponding to the configuration (0, 0, 0). It is also smaller than the bound, so we `AND` this WP with the ones from (0, 0, 0) and (0, 0, 1). We now, again, increase the third element: (1, 0, 1). The corresponding number of instructions is now 40, larger than the bound, so we stop increasing the third element. We then increase the second element: (1, 1, 0). The corresponding number of instructions is now 36, larger than the bound, so we stop increasing the second element. We now increase the first element: (2, 0, 0). The corresponding number of instructions is now 40, again larger than the bound, so we stop increasing the first element. We are now done; the resulting WP consists of three "sub-WPs" and becomes the consequent in the VC.

The function *driver* in `Verification.hs` implements this rather complicated algorithm.

If there is no path whose number of instructions is $\leqslant$ bound, we output an error.

4.2 Examples

4.2.1 `Bounded`

This example squares its first argument:

```
int p (int a0) {
  var x0 = 0;
  var x1 = 0;
  x1 = a0;
  while (x1 > 0) {
    x0 = x0 + a0;
    x1 = x1 - 1;
  }
  return x0;
}
```

The specification is:

$$pre : a_0 \geqslant 0$$
$$post : return \equiv a_0 \times a_0$$

Note that this multiplication is done with addition, e.g. 5×5 is calculated by doing $5+5+5+5+5$.

When we verify this example, the specification is indeed proven correct: `Q.E.D.` When we change the precondition to *true*, the verifier gives an counterexample:

```
Falsifiable. Counter-example:
  a0 = -3 :: SInteger
```

When a_0 is -3, the loop is never entered, so 0 is returned, which is not equal to $-3 \times -3 \equiv 9$.

4.2.2 `Bounded2`

This example just returns its first argument, but uses a loop for this:

```
int p (int a0) {
  var x0 = 0;
  var x1 = 0;
  while (x1 < 1) {
    x0 = x0 + a0;
    x1 = x1 + 1;
  }
  return x0;
}
```

The specification is:

$$pre : true$$
$$post : return \equiv a_0$$

When we verify this example, the specification is indeed proven correct: `Q.E.D.` When we change the postcondition to

$$post : return \equiv 1$$

the output of the verifier becomes:

```
Falsifiable. Counter-example:
  a0 = 2 :: SInteger
```

But when we change the bound to $7 < bound < 20$ and run the verifier again, the specification is proven correct again: `Q.E.D.` This shows the limitations of bounded verification: With the "new" bound, the only possible configuration is (0), i.e. the loop is never entered. But no execution is possible on this path: the condition's guard evaluates (the first time) to $0 < 1 \equiv true$, so every execution should enter the loop at least once (and in this case, just once). As no execution is possible, the specification is "vacuously true": up to 20 (not including) instructions every execution satisfies the specification, as there is no such an execution... When we change the bound to $\geqslant 20$ again, the configuration (1) and, consequently, an execution that does not satisfy the specification, is now possible; when we change the bound to $\leqslant 7$, the number of instructions of no path is $\leqslant$ bound, so an error is given.

4.2.3 `Bounded3, Bounded4, Bounded5`

These examples are variations on the `Bounded` and `Bounded2` examples and consist of multiple, including inner, loops. All can be proven correct and if you change the specification to an invalid one, you will get an counterexample. Verification may take a while, as with multiple loops, including inner ones, and a large bound, many paths are possible and need to be considered.

References

[1] L. Erkök, 2013. *sbv-2.10: SMT Based Verification: Symbolic Haskell theorem prover using SMT solving*. Retrieved April 3, 2013 from Hackage: `http://hackage.haskell.org/package/sbv`.

[2] Microsoft Research, 2013. *Z3 - Home*. Retrieved April 3, 2013 from CodePlex: `http://z3.codeplex.com/`.

[3] S.W.B. Prasetya, 2013. *Project 3: Bytecode Verification Engine*. Retrieved April 3, 2013 from the 'Program Verification' course page: `http://www.cs.uu.nl/docs/vakken/pv/1213/stuffs/PV_P3_1213.pdf`.

[4] Utrecht University, 2009. *Attribute Grammar System*. Retrieved April 3, 2013 from cs.uu.nl's wiki: `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`.