Symbolic Inference of Partial Specifications to Augment Regression Testing

Koen Wermer
Utrecht University
Student number: 3705951

April 22, 2017

1 Introduction

A large part of the costs (50-80% [17]) of software is due to maintenance. A big part of these costs are due to retesting the software to discover newly introduced bugs (also known as regression testing) [17]. We experiment with an approach that may reduce the cost of regression testing by detecting a part of these bugs fully automatically, by generating partial specifications in the form of Hoare triples [1] for both versions of the program (the old version and the new one) and comparing them to eachother. We use a static approach to generate these Hoare triples from source code. We chose Java as the target language for our analysis, because it is widely used and because a lot of the language constructs of Java pose practical challenges that theoretical languages, such as the language considered by Hoare in [1], do not. Because regression testing is a practical problem, it is only natural to perform our experiments on a practical language.

Regression testing, or testing in general, never fully guarantees that the software does not contain errors, so any method that can find errors may find errors that hand written tests do not find. Therefore, our program may increase the number of errors detected without requiring additional work from the programmer. The research question we try to answer is: "Can we detect a decent amount of introduced mistakes without generating too many false positives by comparing partial specifications of the original program to those of the updated version?" The terms "decent" and "too many" are defined by comparing our approach to the tool Daikon [7]. By doing so we'll also find an answer to the question: "Can our approach be used in combination with Daikon to achieve better results?"

1.1 Partial specifications

A way to specify the behaviour of a program (without side effects) is to specify the output value for every possible input value. A practical way of doing so is to abstract over sets of values with the same behaviour, defining these sets using predicates. For example a program *sqrt* that calculates the square root (when we only consider real numbers) could have the specification $\{in \geq 0\}\, sqrt\, \{out = \sqrt{in}\}$ (also known as a Hoare triple [1]) which says that for every input value of at least zero, the output value will equal the square root of the input value. This specification does not say anything about the behaviour of the program if it is executed with an input value lower than zero and it is therefore a partial specification. We could also add the partial specification $\{in < 0\}\, sqrt\, \{out = -1\}$ which states that our implementation of *sqrt* handles negative input by outputting -1. Together, these two partial specification completely describe the behaviour of the program. This makes them useful for catching bugs. If *sqrt* satisfies both specifications it calculates the square root correctly, and if it doesnt satisfy either of the specifications it indicates a bug (in this case, perhaps negative input is handled differently). The partial specifications $\{in = 0\}\, sqrt\, \{out = 0\}$ and $\{in > 0\}\, sqrt\, \{out > 0\}$ are also valid, but not very useful: the identity function also satisfies these specifications. If we were to use these specifications to check for errors and the programmer accidentally returned the input directly (rather than, for example, returning a local variable containing the root) the error would go unnoticed.

In the previous example, $\{in \geq 0\}\, sqrt\, \{out = \sqrt{in}\}$ is objectively more accurate in describing the square root function than $\{in = 0\}\, sqrt\, \{out = 0\}$ and $\{in > 0\}\, sqrt\, \{out > 0\}$ are combined. However, there is no total order for partial specifications in terms of accuracy. This means that, generally speaking, it is hard to quantify 'usefulness' for partial specifications, as it depends on the kind of mistakes the programmers make. Figure 1 shows how different kind of mistakes can be caught by different Hoare triples.

1.2 Regression testing

We want to infer partial specifications from the source code of a program without requiring additional work from the programmer. As a result, the inferred specifications will be useless when testing the original code. However, they can be used when performing regression tests: if refactored code no longer satisfies the original specifications, we know that the refactoring introduced a bug. The amount of bugs that can be found this way is determined by the strength of the inferred specifications. We expect these to be rather weak, as we do not want to burden the programmer and therefore do not have any knowledge about the intended behaviour of the target program. Because of this, our approach is not intended to replace manual regression testing, but rather to accompany it: we might catch bugs that slipped past the manual tests (at no additional cost of the programmer) or achieve the same amount of test coverage with less manually written tests.

1.3 Inferring the specifications

To create partial specifications for a Java program S, we start with a post-condition Q and reason backwards over the list of statements that S consists of, calculating the weakest liberal pre-condition (**wlp**). This is the weakest predicate that guarantees that Q will hold after executing S if we assume that S will terminate. From Q, we've then created the Hoare triple $\{\mathbf{wlp}(S, Q)\}\, S\, \{Q\}$. It is not possible to calculate the **wlp** for arbitrary programs and post-conditions in finite amount of time (if it were, it would imply the existence of an algorithm that solves the halting problem:

$$\mathbf{wlp}(S; assert(false), Q)$$

equals *true* if and only if S does not terminate). Where it does not cause confusion, we'll also use **wlp** to refer to our so-called "approximation" of the **wlp**.

2 The wlp transformer

To answer our research question, we have implemented a **wlp** transformer for Java programs. The transformer is written in the functional programming language Haskell [18]. The transformer is a protoype, as it does not fully support all Java syntax and is not as accurate as could be.

$\{in = 0\}\ sqrt\ \{out = 0\}$
$\{in > 0\}\ sqrt\ \{out > 0\}$
$\{in > 0\}\ sqrt\ \{out < in\}$

```
sqrt(double d)
{
  double rootd;

  rootd = math.sqrt(d);

  //Return the result
  return rootd;
}
```

(a) *Original code*

$\{in = 0\}\ sqrt\ \{out = 0\}$
$\{in > 0\}\ sqrt\ \{out > 0\}$
$\{in > 0\}\ sqrt\ \{out < in\}$

```
sqrt(double d)
{
  double rootd;

  //Specify how negative
  //input is handled
  if(d >= 0)
    rootd = math.sqrt(d);
  else
    rootd = −1;

  //Return the result
  return d;
}
```

(b) *Refactoring 1: The handling of negative input is made explicit, but the wrong variable is returned.*

$\{in = 0\}\ sqrt\ \{out = 0\}$
$\{in > 0\}\ sqrt\ \{out > 0\}$
$\{in > 0\}\ sqrt\ \{out < in\}$

```
sqrt(double d)
{
  double rootd;

  //Specify how negative
  //input is handled
  if(d > 0)
    rootd = math.sqrt(d);
  else
    rootd = −1;

  //Return the result
  return rootd;
}
```

(c) *Refactoring 2: The handling of negative input is made explicit, but the wrong operator is used (> instead of >=).*

$\{in = 0\}\ sqrt\ \{out = 0\}$
$\{in > 0\}\ sqrt\ \{out > 0\}$
$\{in > 0\}\ sqrt\ \{out < in\}$

```
sqrt(double d)
{
  double rootd;

  //Specify how negative
  //input is handled
  if(d == 0)
    rootd = math.sqrt(d);
  else
    rootd = −1;

  //Return the result
  return rootd;
}
```

(d) *Refactoring 3: The handling of negative input is made explicit, but the wrong operator is used (== instead of >=).*

Figure 1: *sqrt satisfies three Hoare triples. The function is refactored to handle negative input explicitly and a bug is introduced in three different ways. Each bug is only caught by one of the Hoare triples.*

As stated in the introduction, we can not calculate the **wlp** in such a way that it is both sound and complete, so the **wlp** we're working with will always be an approximation. This means we had to make a decision: do we want this approximation to be on the weaker side or on the stronger side? We chose the weaker side, based on the following reasoning: in a scenario where we can not determine the **wlp** we can choose between the weakest version, $true$ (indicating the post-condition will always hold), or the strongest version, $false$ (indicating the post-condition will never hold). By definition, the **wlp** assumes termination. This means that if, reasoning backwards from the last statement of the program, we reach $false$ as **wlp** somewhere in the middle, it will continue to stay $false$ and the **wlp** of the program will equal $false$.[1] However, if the **wlp** of the second half of the program is $true$, it can still become stronger when reasoning backwards over the first half if the first half contains (generated) assertions. This means that, even if we can not determine the **wlp** of the second half of a program, we can differentiate between different mutations of the program based on the first half of the code, allowing us to detect changes that otherwise go undetected.
Our algorithm follows the logic of Hoare [1] where applicable. This supplies us with a formula to deal with assignments $x = e$, as long as e does not have side-effects, and procedural composition $S_1; S_2$. In particular, this means we deal with blocks of multiple statements by starting at the last one:

$$\mathbf{wlp}(S_1; S_2, Q) = \mathbf{wlp}(S_1, \mathbf{wlp}(S_2, Q)) \tag{1}$$

The implementation of other statements is described below. The final **wlp** function has a lot more parameters to deal with concepts such as exceptions and breaking of loops. We'll introduce these parameters one at a time and omit parameters that are just passed along. For example, equation 1 is valid, but

assignments
operators
method calls
objects
if-then
if-then-else
conditional expression
while
do
basic for loop
enhanced for loop
empty
assert
switch
break
continue
return
throw
try-catch
try-catch-finally
labels
synchronized
arrays
literal arrays
this
lambda
method reference
instanceof

Figure 2: *Java constructs that are implemented in our prototype. Red colored constructs are not implemented.*

[1]This results from the rule of procedural composition (;) [1]. Procedural composition is associative, therefore equation 1 applies for arbitrary sequences of statements S_1 and S_2. The post-condition $false$ can never be satisfied, therefore if $\mathbf{wlp}(S_2, Q)$ equals $false$ then $\mathbf{wlp}(S_1; S_2, Q)$ must also equal $false$. Any information about S_1 is lost in this transformation, so we want to avoid this situation.

can also be written explicitly as:

$$\begin{gathered}\mathbf{wlp}_{c,mc}(S_1; S_2, Q, B, E, C, R)\\ =\\ \mathbf{wlp}_{c,mc}(S_1, \mathbf{wlp}_{c,mc}(S_2, Q, B, E, C), B, E, C, R)\end{gathered}$$

The additional parameters are needed for dealing with specific language constructs of Java, and will be explained later.
In the current prototype not all Java constructs are implemented, but the most widely used are. Figure 2 gives an overview of the language constructs that are currently supported. Furthermore, our transformer expects code to be preprocessed such that all method and variable names are unique (no shadowing or overloading is allowed) and all member-variables of objects are explicitly referred to with the use of the keyword *this*.

2.1 Unexpected exceptions

Besides generating the **wlp** for a given post-condition, our transformer also generates assertions for statements that may result in an exception. In the current implementation, exceptions are viewed as an inability to satisfy the post-condition. This means that *throw*-statements are equivalent to *assert* $false$. Statements that may throw an unexpected exception can also be rewritten to reflect this behaviour. We currently only implemented this for array access. Figure 3 shows how we can modify source code to make index-out-of-bounds-exceptions explicit.

S	S'

```
int[] a;
int x, n;
...
x = a[n];
```
	```
	int[] a;
	int x, n;
	...
	if(n >= 0 && n < a.length)
	  x = a[n];
	else
	  throw new indexOutOfBoundsException();
	```

Figure 3: *S and S' are equivalent, but S' makes unexpected exceptions thrown by S explicit*

2.2 Expressions

Expressions in Java can have side-effects. This means that they may have an effect on the post-condition. Furthermore, expressions may consist of multiple expressions, combined using operators or methods, each of which can have side-effects. These side effects have to be dealt with in the correct order. For expression that are executed as statements, such as x++, this is easy: x++ is equivalent to the assignment $x = x + 1$. However, x++ is evaluated to x and this expression may be used as part of a larger statement. For example, the

T(n)	= (n, empty)	for all primitive values n
T(v)	= (v, empty)	for all variables v
T(x++)	= (x, x = x + 1)	
T(++x)	= (x + 1, x = x + 1)	
T(x--)	= (x, x = x - 1)	
T(--x)	= (x - 1, x = x - 1)	
T(+e_1)	= T(e_1)	
T($-e_1$)	= ($-T_e(e_1)$, $T_s(e_1)$)	
T($e_1 \oplus e_2$)	= ($v_1 \oplus v_2$, $T_s(v_1 = e_1)$; $T_s(v_2 = e_2)$)	for all binary operators $\oplus$
T(x = e_1)	= (e_1, $T_s(e_1)$; x = $T_e(e_1)$)	
T(x $\oplus$= e_1)	= T(x = x $\oplus$ e_1)	for all assignment operators $\oplus$=
T($e_1 ? e_2 : e_3$)	= (v_1, $if(e_1)\ then\ v_1 = e_2\ else\ v_1 = e_3$)	

Figure 4: *The function T separates expressions from side-effects. By definition, $T(S) = (T_e(S), T_s(S))$ where $T_e(S)$ is the expression part of S and $T_s(S)$ is the side-effect part. A statement S will be translated into $T_s(S)$.*

statement $y = x{+}{+}$ increments x by 1 and assigns the old value of x to y. Simple cases like this are handled directly:

$$\mathbf{wlp}(y = x{+}{+}, Q) = \mathbf{wlp}(y = x; x = x + 1, Q)$$

Expressions that consist of multiple expressions combined with operators, are rewritten to a sequence of simple cases by introducing new variables. For example: if v_1 and v_2 are not referred to by the post-condition Q, we have the equality:

$$\mathbf{wlp}(y = x{+}{+} - x{+}{+}, Q)$$
$$=$$
$$\mathbf{wlp}(v_1 = x{+}{+}; v_2 = x{+}{+}; y = v_1 - v2, Q)$$

The order of the statements is important here. The '$-$' operator first fully evaluates the left argument, so the right argument is affected by potential side effects resulting from evaluating the left argument. The translation to a sequence of assignments makes this explicit. Figure 4 shows the definition of the translation function for expressions consisting of variables, primitive values and operators. How we deal with expressions in the form of methods and expressions that are part of statements (such as if-then-else) will be explained later.

2.3 If-then-else

For the post-condition Q to be satisfied after executing an if-then-else statement, one of the following conditions must be met: the guard is $true$ and executing the then-block will satisfy Q or the guard is $false$ and executing the else-block will satisfy Q. The implication also hold in the reverse direction, meaning that we have the equality:

$$\mathbf{wlp}(if(b)\, S_1\, else\, S_2\,, Q) = (b \wedge \mathbf{wlp}(S_1, Q)) \vee (\neg b \wedge \mathbf{wlp}(S_2, Q))$$

We assume here that evaluation of the guard b does not have any side-effects. This is enforced by introducing a unique variable v_1 in case b does have side-effects, rewriting

$if(b)\ S_1\ else\ S_2$

to

$$v_1 = b; if(v_1)\ S_1\ else\ S_2$$

2.4 Switch and break

Switch statements are rewritten to nested if-then-else statements with one if-then-else statement for every label. Because Java allows control to fall through to the next case if no *break* statement is used, we have to copy the remaining code in the switch block for every case (i.e. all the code following the label). Figure 5 shows an example of a *switch* statement being rewritten. The transformation does not necessarily produce an equivalent program (in fact, the rewritten program might not even compile), but the **wlp** function is defined in such a way to deal with this. Some programming languages, such as C, allow case labels to refer to a statement inside the body of a loop. This is not allowed in Java and therefore the translation to an if-then-else statement can always be made.

```
switch(x)
{
  case 0:   y = 0;
  case 1:   y = 1;
            break;
  default:  y = 42;
}
```

```
if(x == 0)
{
  y = 0;
  y = 1;
  break;
  y = 42;
}
else if(x == 1)
{
  y = 1;
  break;
  y = 42;
}
else
{
  y = 42;
}
```

Figure 5: *Switch statements can be rewritten to a set of nested if-then-else statements.*

For the translation in figure 5 to be correct, our **wlp** function must take into account that the *break* statements inside the if-then-else are meant to stop the execution of the if-then-else statement and not the surrounding loop or switch (if there is even any). In order to make this explicit, we annotate the function with this information: $\mathbf{wlp}(S, Q, B)$ denotes the weakest liberal pre-condition such that Q will hold if S terminates without encountering a *break* statement, and the predicate B will hold if a *break* is encountered. If S is a switch statement and S^{ite} is the translation to an if-then-else block, we get the equality

$$\mathbf{wlp}(S, Q, B) = \mathbf{wlp}(S^{ite}, Q, Q)$$

We can now also define the **wlp** of a *break* statement:

$$\mathbf{wlp}(break, Q, B) = B$$

2.5 Exceptions and try-catch-finally

To handle a try-catch-finally statement, we use a similar approach to the one we used to deal with *break* in order to deal with *throw*. We treat the execution of a try-catch-finally statement as an execution of the code within the try-block and the (potential) finally-block, but also keep track of the catch-blocks and whether or not a finally block was available. *throw* statements simply transfer the control to the matching catch or finally block, if available. In order to define the calculation, we have to further annotate the **wlp** function with a structure c that contains the necessary information. c consists of an (ordered) list of catch-blocks and a boolean value indicating whether or not a finally-block is present. A catch-block consists of a block of statements and the type of the exception that can be caught by the block.

If S is a try-catch-finally statement as defined in figure 6, the rule to dealing with try-catch-finally becomes this:

$$\mathbf{wlp}_c(S, Q, B) = \mathbf{wlp}_{c'}(S_0, \mathbf{wlp}_c(S_f, Q, B), B)$$

where c' contains the list of catch-blocks and the value *true* to indicate a finally block exists. If there is no finally-block, the rule becomes

$$\mathbf{wlp}_c(S, Q, B) = \mathbf{wlp}_{(c',c)}(S_0, Q, B)$$

where c' contains the list of catch-blocks and the value $false$. Note that we tuple the new catch information c' with the old information c, essentially creating a stack of catch information (c might be a tuple itself, forming the rest of the stack). This is needed in case S_0 throws an exception that can not be caught by any of the catch blocks. In this case the exception can still be caught by an encompassing try block. If there is a finally block present, this is not needed because exceptions that are not caught by any of the catch blocks are caught by the finally block.
Because an exception inside of a try block prevents the rest of the try block from being executed, we need to keep track of what the original post-condition at the end of the try block. We need to further extend the parameters of the **wlp** function with an additional predicate E: $\mathbf{wlp}_c(S, Q, B, E)$ calculates the **wlp** over S where B and c are as previously defined, Q is the post-condition that needs to be satisfied if the rest of the block that S is part of will be executed, and E is the post-condition that needs to be satisfied if the rest of the block

```
S
try
{
   $S_0$;
}
catch($E_1$ $e_1$)
{
   $S_1$;
}
catch($E_2$ $e_2$)
{
   $S_2$;
}
...
catch($E_n$ $e_n$)
{
   $S_n$;
}
finally
{
   $S_f$;
}
```

Figure 6: *A general try-catch-finally statement with n catch blocks.*

will not be executed. Most previously defined rules do not affect E and just pass it along (as they do with B and c). For example, the rule for composition becomes

$$\mathbf{wlp}_c(S_1; S_2, Q, B, E) = \mathbf{wlp}_c(S_1, \mathbf{wlp}_c(S_2, Q, B, E), B, E)$$

However, we need to update the rules for try-catch and try-catch-finally:

$$\mathbf{wlp}_c(S, Q, B, E) = \mathbf{wlp}_{c'}(S_0, \mathbf{wlp}_c(S_f, Q, B, E), B, \mathbf{wlp}_c(S_f, Q, B, E))$$

in case there is a finally block and

$$\mathbf{wlp}_c(S, Q, B, E) = \mathbf{wlp}_{(c',c)}(S_0, Q, B, Q)$$

if there is none.

We now have four separate possibilities to deal with when a *throw* statement occurs:

1. If the exception is caught by the n^{th} catch block of c, we can use the rule

   $$\mathbf{wlp}_{(c,c')}(\mathit{throw}\, e, Q, B, E) = \mathbf{wlp}_{c'}(S_n, E, B, E)$$

   which comes down to inserting S_n in place of the *throw* statement and ignoring the rest of the code in the block.

2. If the exception is not caught by a catch block of c, but c indicates there is a finally block, we can simply ignore the error and the rest of the block. The rule for try-catch-finally has already taken the code of the finally block into account. In this case we can use the rule:

   $$\mathbf{wlp}_{(c,c')}(\mathit{throw}\, e, Q, B, E) = E$$

3. If the exception is not caught by a catch block of c and there is no finally block, we have to throw the exception. However, this try-catch statement might be part of another try block, so we let the next block deal with it:

   $$\mathbf{wlp}_{(c,c')}(\mathit{throw}\, e, Q, B, E) = \mathbf{wlp}_{c'}(\mathit{throw}\, e, Q, B, E)$$

4. The throw statement might not be part of a try block at all. We use () to indicate this. In this case we know that the exception will definitely be thrown, which in our current implementation comes down to asserting $false$:

   $$\mathbf{wlp}_{()}(\mathit{throw}\, e, Q, B, E) = false$$

Until now, we have assumed that we can determine whether or not an exception is caught by a catch block. This is not always the case. The type of exceptions, just like the type of other Java objects, can change dynamically. Because our analysis is static, we have to make an approximation, which causes imprecision. Furthermore, our current implementation does not take (user defined) class hierarchy into account. If an exception e of type T is thrown, we only look at catch blocks that match the type T exactly, or that catch exceptions of the most general type *Exception*.

2.6 Loops

Since we do not want to burden the programmer with the obligation to specify a meaningful invariant for loops or recursive calls, we choose an unrolling approach (similar to the translation from loops as described in [3]). For a while loop $\mathit{while}(b, n, S)$, where b is the guard, S is the body and n is the maximum number of executions of the body that we want to include in our analysis, we can approximate the **wlp** with the two (simplified) rules

$$\{\neg b \implies Q\}\, \mathit{while}(b, 0, S)\, \{Q\}$$

and

$$\{\neg b \implies Q \wedge b \implies \mathbf{wlp}(S; \mathit{while}(b, n-1, S), Q)\}\, \mathit{while}(b, n, S)\, \{Q\}$$

These rules assume that b has no side-effect. As discussed earlier, Java expressions can have (multiple) side effects. In order to deal with this, we use the same approach as when dealing with side effects of sub-expressions: we introduce a new variable for the guard. The guard is evaluated at the start of every iteration of the loop, with side effects occurring even if the guard evaluates to false. Figure 7 shows how we can transform the while loop to reflect this behaviour. For-loops and do-loops can be easily rewritten to while-loops, and from there transformed to a while-loop with no side-effects in the guard. Enhanced for-loops are currently not supported, but they can also be rewritten as they are syntactic sugar for a combination of a while loop in combination with an *Iterator* object.

```
while(b)
{
  $S_1$;
}
```

```
$v_1$ = b;
while($v_1$)
{
  $S_1$;
  $v_1$ = b;
}
```

Figure 7: *A transformation of a while loop in order to deal with possible side-effects of the guard b.*

The next step is to extend our two rules for handling while loops without side effects in the guard, to reflect the additional information that is passed. For the base case, the body is completely ignored, so we do not have to deal with *break* statements. We also do not have to deal with try-catch, as only the guard is evaluated, and it has no side effects (therefore no exceptions are thrown). The rule is therefore

$$\mathit{wlp}_c(\mathit{while}(b, 0, S), Q, B, E) = \neg b \implies Q$$

In case we do execute the body, we have to pass try-catch information to the calculation that handles the body, as well as supplying the information that *break* statements finish the execution of the loop. This gives us the rule

$$\begin{gathered}\mathbf{wlp}_c(\mathit{while}(b, n, S), Q, B, E) \\ = \\ (\neg b \implies Q) \wedge (b \implies \mathbf{wlp}_c(S; \mathit{while}(b, n-1, S), Q, Q, E))\end{gathered}$$

We can now use the same rule for dealing with *break* that we use in the context of switch statements:

$$\mathbf{wlp}_c(\mathit{break}, Q, B, E) = B$$

A *continue* statement indicates that the current iteration is finished. We can not express this with the current parameters, so we introduce a new parameter C. When unrolling a loop, we pass the information that *continue* refers to the next iteration:

$$\begin{gathered}\mathbf{wlp}_c(\mathit{while}(b, n, S), Q, B, E, C) \\ = \\ (\neg b \implies Q) \wedge (b \implies \\ \mathbf{wlp}_c(S; \mathit{while}(b, n-1, S), Q, Q, E, \mathbf{wlp}_c(\mathit{while}(b, n-1, S), Q, Q, E, C)))\end{gathered}$$

The rule for *continue* then becomes rather simple, similar to *break*:

$$\mathbf{wlp}_c(\mathit{continue}, Q, B, E, C) = C$$

As discussed earlier we chose for the weak approximation, using implications rather than conjunctions. If a specific execution of a loop iterates more than a specified number n times we simply assume that the post-condition will hold. This means our analysis is not sound: the pre-condition that is inferred only guarantees that the post-condition will hold for executions that execute the body of the loop at most n times. It is, however, complete: if a loop will satisfy the post-condition after execution, than the pre-condition holds before execution. In our experiments we set the maximum number of iterations n to 1.

2.7 Method calls

For method calls, the idea is fairly similar to loops: we take the execution of the body into consideration if and only if it does not go over the set amount of recursive calls. Otherwise, we just assume the call will always satisfy the post-condition. Since we set the maximum number of iteration to 1, our approximation of the **wlp** for recursive calls equals *true*. To deal with *return*, we add an additional parameter R. Figure 8 shows how we can inline a call to a static method without a return value. We assume here that the parameters do not have side-effects. In case parameters do have side-effects, we get rid of them by introducing variables, the same way we did for loops. If S^{inline} is the body S of the method m that is transformed in this way, the rule

$$\mathbf{wlp}_{c,mc}(m(p_1, .., p_n), Q, B, E, C, R) = \mathbf{wlp}_{c,mc'}(S^{inline}, Q, B, E, C, Q)$$

applies if we are not on the recursion limit. Here $p_1, .., p_n$ are the parameters of m and mc is a dictionary that denotes the number of calls that still can be made for each method in the program. mc' is the updated version of mc, where we subtract 1 from the the value for m. Because we pass this information for every method in the program, we also deal with mutually recursive calls correctly. If we are on the recursion limit (i.e. mc holds the information that we no longer want to look at the body of m), we have the rule

$$\mathbf{wlp}_{c,mc}(m(p_1, .., p_n), Q, B, E, C, R) = true$$

We can now define the **wlp** of *return* statements:

$$\mathbf{wlp}_{c,mc}(return, Q, B, E, C, R) = R$$

If the method returns a value, we also introduce a new unique variable for the return value. *return* statements are treated as assignments to this variable, as well as referring to the return condition R. Figure 9 shows how this transformation is done. The rules for calculating the **wlp** stay the same. In the case that we reached the recursion limit, we do not look at the body and therefore can not produce a return value. However, in that case the **wlp** is set to *true*, so it does not matter what value we substitute for the return variable of the function, as the expression *true* does not depend on that variable.

```
m(e1,..,en);                    //Assign the parameters
                                p1 = e1;
static void m(p1,..,pn)         ...
{                               pn = en;
  if(b)
    return;                     //Execute the body
  S;                            if(b)
}                                 return;
                                S;
```

Figure 8: *Inlining a call to a static void method.*

```
x = m(e1,..,en);

static int m(p1,..,pn)
{
  S;
  return y;
}
```

```
//Assign the parameters
p1 = e1;
...
pn = en;

//Execute the body, modifying
//any return statements
S;
v1 = y;
return;

//Execute the encompassing
//statement
x = v1;
```

Figure 9: *Inlining a call to a static method that returns a value.*

So far we've only considered static methods. Because we pre-process the code to make references to the current instance explicit using *this*, we can easily extend our approach to calls on objects by substituting the object for every occurrence of *this*. Figure 10 shows an example of this. We do get some imprecision, however, due to the fact that types of Java objects can change dynamically. If there are multiple possibilities, we choose the method that matches the originally declared type of the object.

```
x = o.m(e1,..,en);

class O
{
  int y;
  int m(p1,..,pn)
  {
    S1;
    this.y = z;
    S2;
    return this.y;
  }
}
```

```
//Assign the parameters
p1 = e1;
...
pn = en;

//Execute the body, modifying
//return and this
S1;
o.y = z;
S2;
v1 = o.y;
return;

//Execute the encompassing
//statement
x = v1;
```

Figure 10: *Inlining a call to an instance method that returns a value.*

```
public static void main()
{
  P origin = new P(1, 0);
  origin.px -= 1;
}

public class P
{
  public float px, py;

  public P(float x, float y)
  {
    this.px = x;
    this.py = y;
  }
}
```

```
public static void main()
{
  //Introduce a name
  P origin = object1;

  //Assign the parameters
  x = 1;
  y = 0;

  //Constructor code
  object1.px = x;
  object1.py = y;

  //Rest of main code
  origin.px -= 1;
}
```

Figure 11: *Inlining of constructor calls: On the left we have the original code, and on the right we have the main function how it is handled by the transformer.*

2.8 Objects

When creating a new object, we give it a unique name. We then inline the constructor, replacing every *this*-keyword by the introduced name (as stated before, we assume that references to the current object are made explicit using *this* during pre-processing). Figure 11 gives an example of such a transformation.

We also have to add an additional rule for handling assignments to fields of objects. The original rule,

$$\mathbf{wlp}(x = e, Q) = Q[e/x]$$

where $Q[e/x]$ denotes substitution of every free occurrence of x in Q by e, also hold for class types. In this case e can be thought of as an expression evaluating to an integer, representing the location of the object in memory. The tricky part is dealing with assignments of the form $o.x = e$. This does not only affect every occurrence of $o.x$ in Q, but also every occurrence of $p.x$ where p is any expression that points to the same location as o. Because we do not know what pointers will evaluate to o, we have to consider both possibilities using an if-then-else-expression:

$$\mathbf{wlp}(o.x = e, Q) = Q[if\ o = p\ then\ e\ else\ p.x/p.x]_{\forall p \in Exp}$$

where Exp is the set of all expressions, o is an object and x is a field of o. In practice, we only apply the substitution for those expressions p in the post-condition that have the same type as o.

2.9 Limitations

There are some limitations to this approach in comparison to dynamic approaches. The most obvious one is the inability to reason about library methods. Since we calculate the **wlp** statically and do not have access to the source code of library methods, we do not know how a library method call will affect the post-condition. In our prototype we just assume the a library method call will satisfy the post-condition (however, there are ways to make the analysis more accurate by considering the parameters of the method call, see 'future work'). Another limitation is caused by the way we deal with loops and recursion. Sadly, there is no way to calculate the sound and complete **wlp** fully automatically (without an invariant supplied by the programmer). A side effect of our solution is that the inferred invariant is sensitive to changes in the control flow: the order in which loops are executed and methods are called affects the **wlp**, even when all statements commute with eachother. An example is given below.

S

```
for(int i = 0; i <= 5; i++)
{
  if(i == 5)
    assert false;
}
```

S'

```
for(int i = 5; i >= 0; i--)
{
  if(i == 5)
    assert false;
}
```

Figure 12: *A mutation that would result in a false positive.*

In the above example, $\textbf{wlp}(S, Q)$ will evaluate to $true$, while $\textbf{wlp}(S', Q)$ will evaluate to $false$ (assuming we unroll loops less than 6 times). Therefore $\textbf{wlp}(S, Q) \implies \textbf{wlp}(S', Q)$ will evaluate to $false$ and we might conclude that S' introduces a new bug even though S and S' are equivalent.

3 Research methodology

The **wlp** transformer is used for regression testing in the following way. We consider a program S and a modified version of the program, S'. S' may refactor some of the code of S, fix some bugs and add additional features, but the main assumption is that methods that have the same name and parameters are intended to behave to same (except for possibly some bug fixes). Using the **wlp** transformer and a post-condition Q, we then generate, for every method m in S that has a matching method m' in S', the expression

$$\textbf{wlp}(m, Q) \implies \textbf{wlp}(m', Q) \tag{2}$$

To determine whether or not this expression is true, we use the theorem prover Z3 [22]. If Z3 can prove the generated expression to be valid for all such pairs m and m', we say that S' did not introduce any errors (as far as we can tell), otherwise S' likely introduced an error. The reason we only consider the implication in one direction (rather than program equivalency) is that our approach should be useful for regression testing. Input that does not violate the

Construct	BaseSecantSolver	GradientFunction	Iterator
If-then(-else)	15	0	6
Switch	3	0	0
Assert	0	0	0
Loops	1	2	1
Break/continue	7	0	0
Try-catch(-finally)	0	0	2
Throw	12	0	6
Array access	0	5	3

Table 1: *The number of language constructs used for each test class.*

post-condition or any generated constraints (as described in section 2.1) should still not do so after the code has been modified (this is enforced by equation 2). However, if certain input used to violate the post-condition, but no longer does, then this might just be a bug fix. For example, $\mathbf{wlp}(m', Q) \implies \mathbf{wlp}(m, Q)$ might be $false$ as the result of a bug that caused an unexpected exception to be thrown in S that was fixed in S'. Equation 2 gives the programmer the freedom to fix these kinds of bugs, while still enforcing previously valid input to still be valid.

We test this approach by performing a mutation test (to test how many errors we can detects) and a false positives test (to test how often we generate a false positive).

3.1 Mutation test

To test the ability to detect errors with our approach we use the Major mutation tool [19]. Major can create a large number of mutants of the target program and output the mutated source code. We tested our prototype on three real-world classes from the Apache Commons Mathematics Library [23]: *BaseSecantSolver*, *Iterator* and *GradientFunction*. We selected the classes based on the language constructs they use. *BaseSecantSolver* is an abstract class[2] that uses a multiple switch statements. *Iterator* is a local class that uses try-catch-statements and relatively many throw statements (6 throw statements in 63 lines of code). *GradientFunction* is the only class of these test cases that uses arrays (this is note-worthy, as the only type of unexpected exceptions implemented by our prototype are index-out-of-bounds-exceptions, as discussed in section 2.1). Table 1 shows the constructs that are used per class, and how many of them are used. Since loops are all translated to while-loops, as discussed in section 2.6, we don't differentiate between different kind of loops in the table. The most basic constructs (assignments, operators and method calls) are omitted from the table, but are used in all of the test classes.

[2]*BaseSecantSolver* is abstract, but it does have an implementation for most of the methods, so we can analyze those methods.

Construct	2D_to_1D	BST	Fibonacci	MinsMaxs	Normalizer	Stack	Vector	Vector01Gen
If-then(-else)	0	7	0	4	3	4	1	1
Switch	0	0	0	0	0	1	0	0
Assert	0	0	0	0	0	0	0	0
Loops	7	1	1	4	3	2	3	2
Break/continue	0	0	0	0	0	5	1	0
Try-catch(-finally)	0	0	0	0	0	0	0	0
Throw	0	0	0	0	3	0	1	0
Array access	5	0	0	16	6	4	6	1

Table 2: *The number of language constructs used for each class used in the false positive test.*

3.2 False positives test

To test how many false positives we generate, we asked two programmers to makes small refactorings to various classes in such a way that every method in the refactoring is equivalent to the original version of the method (although methods may be added or removed). We used 8 test classes and a total of 19 equivalent mutations of these classes for this test. Table **??** shows the constructs that are used per class of the 8 original classes. Tablekind of mutations shows in what way these classes have been refactored. Note that some classes have multiple mutations, and some only have one.

3.3 Baseline test

As a baseline test, we compare our method to the dynamic invariant inference tool Daikon [7]. Daikon can also be used to infer properties of Java programs without requiring additional information, so it makes sense to make this comparison. We use Daikon as follows: For a given test class and method, we use the test generator T3 [20] to generate a test suite that covers all reachable branches. We then run Daikon with the origial class and the test suite to generate a set of invariants (i.e. properties that hold for every execution in the test suite). We then run Daikon again, using the test suite and the invariant file, on the mutations of the original class to check if the invariants still hold. We do this for every test class and every method and count the number of mutations in which we find an error (i.e. at least one invariant does not hold).
In order to use T3 and Daikon, we had to make small modifications to the source code of the test classes, such as adding a public constructor in order to instantiate member variables, and specifying which implementations of abstract classes are used. When doing so, we made sure all branches are covered by the test suite in order to make the comparison fair. We also eliminated the addition mutations that resulted from Major mutating the added code.

Test Class	Mutation
2D_to_1D	The content of the array is only stored as a 2D array. The method *convert* (that converts the 2D array to 1D and stores it as a member variable) is removed.
	The counter k in a nested for-loop (to loop through a 2D array) that denotes we passed k elements is removed. Instead this number is calculated from the counters of the inner and outer for-loops.
BST	The nodes of the binary search tree no longer contain a reference to their parent.
Fibonacci	Instead of printing the numbers 0 and 1, followed by a loop that prints the next n numbers in the Fibonacci sequence, the loop has been rewritten to iterate twice more and the other 2 print statements are removed.
MinsMaxs	Some <= and >= operators are replaced to improve efficiency.
	Two for-loops that loop through an array are merged into one.
	A nested for-loop that loops through each element of each row in a 2D array is replaced by a nested for-loop that loops through each element of each column.
Normalizer	An if-then-else statement that throws an exception in the else-block is changed to an if statement that throws the exception.
	The original class only has one method to normalize all values in a 2D array of doubles. A method is added to normalize a single double value and the other method now calls this one.
	A while-loop is rewritten to a for-loop.
	The exceptional control flow is rerouted by throwing a different exception and catching it.
Stack	An if statement that returns *true* is replaced by directly returning the guard.
	The original class has two constructors with code-duplication. This is fixed by calling one constructor from the other using the keyword *this*.
	A variable denoting the size of the stack is removed from the data structure. The length of the underlying array is used instead.
Vector	A *return* inside a switch-block is replaced with a *break*. The *return* is moved to after the switch-block.
	A fall-through case in a switch-block is replaced with a recursive call.
	The cases in the switch-block are put in a different order.
Vectors01Generator	A for-loop that only iterated twice is replaced by twice the statements in the body.
	Instead of printing the output line by line in a loop, a string is constructed in the loop instead and then printed after the loop.

Table 3: *The equivalent mutations for each class used in the false positive test.*

Heuristic	PC for void methods	PC for methods that return a primitive type	PC for methods that return a reference type
heur0	*true*	*true*	*true*
heur1	*true*	return value = v_{ret}	*true*
heur2	*true*	*true*	return value != *null*
heur3	*true*	return value = v_{ret}	return value != *null*

Table 4: *The definitions of the different heuristics. v_{ret} is an introduced variable that appears as a free variable in the **wlp**. PC stands for post-condition.*

4 Results

We ran the tests with different post-conditions, with post-conditions depending on the return type of the method. Table 4 summarizes the different heuristics we tried. In these post-conditions, v_{ret} is an introduced variable that appears as a free variable in the **wlp** of both the mutation and the original source code. Using, for example, return value = v_{ret} as a post-condition can therefore be interpreted as demanding the return value of the method in the mutated class to be equal to the return value of the method in the original class.

The three test classes we used for the mutation test are *BaseSecantSolver*, *GradientFunction* and *Iterator*. *BaseSecantSolver* contains one large method that tries to numerically approximate roots of a given function (i.e. values that are mapped to zero). Most part of this function is a big while-loop. It also contains a few small methods that do not contain loops. *GradientFunction* contains one method to calculate the gradient of a function. The input and output are represented as arrays of doubles. *Iterator* is a local class of *OpenIntToDoubleHashMap* and contains methods to get the key and value of the current entry and to advance to the next entry in the hash map. Table 5 shows some statistics about the used test classes. The first three classes in the table are the classes used for the mutation test. The other classes are used in the false positives test.

Table 6 shows, for every test class, the number of mutations and the number of errors detected by each of our heuristics and by Daikon. Because T3 generates test suites randomly, the invariants Daikon infers may be different every time the test is ran. This also means that the mutations that are detected may differ every run. Because of this we ran the test 3 times for every class. Every time Daikon detected the same mutations, except for one test run on the Iterator class where it detected one additional error that was not detected by any of our heuristics (this is not shown in the tables).
It can be seen from table 6 that the percentage of errors detected by our heuristics is by far the lowest for *BaseSecantSolver*. This is because the main part of the code is one big while loop for which, generally speaking, the body is executed multiple times (as it is meant to numerically approximate roots of a function by iterating). Since we only considers executions that execute the body as most once, we don't find any errors by analyzing this method. The other methods of the class are smaller, and we do in fact detect errors by analyzing those methods. The fact that we detect a lot

Test Class	Lines of code	Number of constructors	Number of methods	McCabe complexity per method (min-max-average)
BaseSecantSolver	185	3	7	1 - 33 - 5.0
GradientFunction	23	1	1	1 - 3 - 2.0
Iterator	63	1	4	1 - 5 - 2.8
2D_to_1D	64	1	5	1 - 3 - 2.2
BST	115	1	7	1 - 6 - 2.0
Fibonacci	19	0	1	2 - 2 - 2
MinsMaxs	26	0	1	9 - 9 - 9
Normalizer	22	0	1	6 - 6 - 6
Stack	108	2	11	1 - 7 - 1.8
Vector	39	1	3	1 - 8 - 3.0
Vectors01Generator	33	0	3	1 - 3 - 2.0

Table 5: *Some statistics for the used test classes.*

more errors when taking the output of functions into account (22 instead of 2) can be explained by the fact that the smaller functions are easy to analyze (they don't contain any loops or recursion) and three of them return a primitive value. None of the functions of $BaseSecantSolver$ return an object, so heuristic 2 does not increase the number of errors found relative to heuristic 0.

Table 6 also shows the number of errors detected for the false positives test. Note that the false positive test does not refer to a single test class, but is an aggregation over all 8 test classes we used for this test. Daikon infers invariants that are true for all of the cases in the test suite. Since we use the same test suite to check the invariants on the mutations, Daikon will never return a false positive. However, this does not mean that the invariants inferred by Daikon for the given test suite are actual invariants (when taking all possible inputs into account). How accurate Daikon is in this aspect is shown in [21].
It can be seen from table 6 that we generate one false positive when we take the return value of methods that return an object into account. This is caused by an equivalent mutation of the $Vector$ class that changes the control flow of a method (that happens to return an object), making it recursive. As shown in figure 12 our analysis is sensitive to these kind of changes.

Table 7 shows how many errors we can detect if we use both Daikon and one of our heuristics and count every mutation that is labeled erroneous by either of the analyses. It can be seen from the table in comparison to table 6 that the set of errors detected by Daikon and the set of errors detected by our prototype are almost mutually exclusive.

Table 8 shows the size of the **wlp**, which we define as the number of literal values and variables occurring in the expression. It also shows the average run-time per mutation for the mutation test (this includes the time Z3 takes to check the expression). It shows these numbers for both $heur0$ and $heur3$.

[3]Using an AMD FX-6300 six core processor.

Test	Total mutations	heur0	heur1	heur2	heur3	Daikon
BaseSecantSolver	207	2	22	2	22	3
GradientFunction	26	4	4	6	6	5
Iterator	43	13	20	13	20	7
False positives	19	0	0	1	1	0

Table 6: *The total number of mutations for each test and the number of mutations that were determined to be erroneous by each of our heuristics compared to Daikon.*

Test	Total mutations	heur0 & Daikon	heur1 & Daikon	heur2 & Daikon	heur3 & Daikon
BaseSecantSolver	207	5	25	5	25
GradientFunction	26	8	8	10	10
Iterator	43	18	25	18	25
False positives	19	0	0	1	1

Table 7: *The total number of mutations for each test and the number of mutations that were determined to be faulty by each of our heuristics in combination with Daikon.*

Test	size of the **wlp** (heur0)	average run-time (heur0)	size of the **wlp** (heur3)	average run-time (heur3)
BaseSecantSolver	49	92ms	56	93ms
GradientFunction	26	15ms	26	16ms
Iterator	133	63ms	139	66ms

Table 8: *The size of the* ***wlp*** *(summed over all methods in the class) and the average run-time*[3]*for checking a mutation.*

5 Related work

Different approaches have been suggested to balance preciseness and cost of computing a weakest pre-condition. Originally, Hoare introduced Hoare logic based on axioms that describe which Hoare triples are valid [1]. However, the language that is considered here is very impractical, as it has no exceptions or methods and requires the programmer to explicitly annotate loops with invariants. [2] extends this logic by adding an *abort* command and furthermore bases the rules on operational semantics of the language (rather than regarding the rules themselves as axioms). However, it does not deal with the aforementioned impractical aspects of the language. [3] presents a way to generate verification conditions for java by compiling the java program to a primitive guarded-command language, losing some precision in the process. This guarded-command language is then used to calculate the verification conditions. To keep flexibility for experimenting with different kinds of translations from java to the primitive language (balancing trade-offs in different ways), the authors introduced a sugared guarded-command language. The first translation step gets rid of a lot of the complexities of java, but without loss of information (so it does not need to be flexible). The second step is the one that actually needs to be flexible, and is made easier by the first step. More recently (in 2008, to be specific), the Boogie language was created for the purpose of being an intermediate language for program verification [4][5].
Another approach is presented in [6]. Rather than using an intermediate language, programs are simplified by unrolling loops and recursion a small number of times (for while loops, this can be done by replacing the loop with a finite amount of if statements with the same guard and body as the while loop). For dealing with method calls, specifications of methods are over-approximated. This implies that if the resulting verification condition is verified (that is, it's negation is proven to be unsatisfiable), then it is valid even with the true specifications of all methods involved. If the negation is satisfiable, satisfying it produces a counter-example (in the form of input) for the program with the approximated method specifications. The program can then be ran with the counter-example as input to find out whether it's also a counter-example to the program with the actual code of the methods. If it turns out not to be, then this disproves the specification for some method and the specification can be refined: during execution of the counter-example the input and output of every method call is compared to the approximated specification of that method and if the pair of input and output does not match the specification, the pair can be used to make a better approximation of the specification. This process is repeated until the verification condition is either validated or disproven. Because all loops are capped at a fixed number of iterations, termination is guaranteed. However, this means that verified conditions are not necessarily valid in the original program.

Invariants are properties of a program that do not depend on the input. Invariants often take the form of predicates that are always true on a specific line in the program code. For example, a while loop implementing selection sort has the property that at the end of the body the first i elements of the list are sorted (where i is the number of iterations performed so far). Invariants play an important role in creating partial specifications, as they can be used to compute the weakest pre-condition of a loop [1]. A popular tool to dynamically infer likely invariants is Daikon [7]. Daikon checks for 75 different invariants, but can also be extended to check for other invariants [7]. Because invariants are inferred dynamically they are not guaranteed to hold for all possible executions. By dividing input into clusters of values that follow the same branch in the program, Daikon may be able to find more valid invariants [8]. Another approach for differentiating between branches is presented in [9]. The authors specify a method to infer invariants dynamically for different methods and different branches within the methods. Comments are used to identify these branches. These comments, together with the inferred invariants, may then be presented to the programmer for inspection. The invariants are limited to different kinds of simple invariants. Each kind of invariant had its own lattice defined on it, so that resulting invariants of multiple executions can be joined, thus dynamically inferring the invariants. Because of the simplicity of the invariants, they can be easily verified by the programmer. However, by performing a mutation test the authors show that the invariants are strong enough to catch errors and in combination with invariants produced by Daikon can catch more errors than Daikon by itself [9]. [10] also presents a way to infer invariants (again by fitting to previously defined invariant patterns), but in contrast to the previously mentioned papers it focusses on temporal properties, which can be useful especially when dealing with concurrency. Algebraic equations form another type of invariant. [11] is concerned with inferring this kind of invariant. More specifically, equations concerning logged events are inferred in order to rewrite the event log in such a way that it becomes simpler, but the new series of events still lead to the same final state. Other properties of a program that may be viewed as a kind of invariant and can be inferred dynamically are finite state automata [12].

Computing the weakest pre-condition for a loop with a given post-condition statically using Hoare logic involves specifying an invariant for the loop [1]. For the approach used in [9] this does not apply, as dynamic inference is used. The authors of [3] simply use either *true* or an invariant specified by the programmer, as generating invariants is not the topic of their research. They do, however, state that there are a lot of different approaches for refining invariants in the literature and claim that their program is flexible enough to easily implement such an approach. One approach is presented in [13]. This approach is based on the observation that invariants are a weaker version of the post-condition. Using various heuristics the authors systematically make the post-condition weaker and check if it's an invariant. [14] essentially uses the same approach but in the opposite direction. Their program starts with a valid invariant which is made stronger until the verification condition is satisfied.

A way to combine (dynamically) inferred partial specifications with testing is presented in [15]. This method is based on inferring partial specifications dynamically from executing a unit test suite. The number of tests is extended by iteratively selecting useful tests from a set of tests that are automatically generated using Jtest [16]. A new test may result in violating the specifications inferred so far. The authors argue that when this happens, the new test is likely to cover a feature of the program was not covered by the original test suite, so when an automatically generated test violates the current specifications the programmer may decide to add it to the test suite.

6 Conclusion

We have shown that partial specifications in the form of Hoare triples can be used to automatically catch bugs that are introduced into Java code. To show this, we have built a prototype wlp-transformer and performed a mutation test. Our transformer, in contrast to the transformer introduced in [3], is applied directly on Java source code. Even though the wlp-transformer is not as accurate as can be and does not take all the available information into account (see 'Future Work'), we can use it to detect a significant amount of mutations that are not detected by Daikon. We have also shown that the number of false positives generated by using this method is limited, although mapping out the exact trade-off between true positives and false positives requires more research.

6.1 Future work

As mentioned earlier, our prototype does not support all of the Java constructs. These constructs can still be added to the program to support a wider range of Java programs. Our prototype is coded in such a way that it is easy to add support for Java constructs that are not yet covered. Currently, variable shadowing and referring to member variables without an explicit *this*-statement is not supported and pre-processing for the experiment was done by hand. It should be relatively easy to write a pre-processor that renames variables to a unique name and adds explicit *this*-statements where required.

So far, we've only considered the return value of methods when deciding on a post-condition. However, creating a post-condition that depends on the entire state of the program may be better at catching errors. If we consider the entire state of the program, this gives possibilities for a lot of potential post-conditions. On top of that, there's a lot of experimentation that can be done with inserting assertions into the target program. As mentioned, we did this for accessing arrays, in such a way that the generated assertion states that the array access will not throw an error. We can implement the other built-in exceptions in a similar way, but there are a lot more possibilities. For example, if the target program contains loops with a guard of the form $i<n$, it might be fruitful to assert $i{==}n$ right after the loop. A guard like this can be used to execute the body of the loop n times and if this was indeed the programmer's intention, the assertion should hold for both the original program and its mutation. If it does not hold anymore for the mutation, this might indicate that i is being incremented too often. Assuming this is a mistake that is being made a fairly often, this could be a nice thing to implement. Whether or not this specific heuristic is actually useful is pure speculation at this point in time, but with this kind of reasoning one could come up with many heuristics that may be worth testing.

In our prototype we do not differentiate between different types of exceptions and instead just treat every thrown exception as an execution of the program that does not satisfy the post-condition. It is possible to make the analysis more precise by differentiating between normal termination and erronous termination as described in [3]. Furthermore, we can extend the program to differentiate between expected exceptions and unexpected built-in exceptions.

In the current implementation we make no assumptions about library methods. We can make this approach more precise by reasoning about the variables that the function has access to. For example, if we declare an *int* x and import a library method *foo*, we can conclude that the Hoare triple $\{x = 5\}\, foo(..)\, \{x = 5\}$ is valid, because *foo* has no way of changing the value of x. Even if x is used as an argument in the call to *foo*, it would be copied by value, not by reference, and the variable reference would not be in scope in the method body. If x were to be a field of an object o instead, we can infer $\{o.x = 5\}\, foo(..)\, \{o.x = 5\}$ as long as o is not passed as an argument to *foo* and no object that *foo* has access to has a reference to o.

The idea behind the experiment is to aid the programmer without requiring additional work. To achieve this, ultimately our program will have to be integrated into an IDE so that running the analysis will be no hassle (and the analysis might even be running automatically while programming). To further aid the programmer, Z3 can be asked to find parameters that form a counter-example to the inferred formula when a bug is found [22]. This makes it easier for the programmer to identify false positives as well as to identify the cause of the bug if it is indeed a bug.

References

[1] C. A. R. Hoare *An Axiomatic Basis for Computer Programming*, Communications of the ACM, vol. 12, pp. 576-583, 1969.

[2] Peter V. Homeier, David F. Martin *A mechanically verified verification condition generator*, The Computer Journal, Vol. 38, No. 2, pp. 131-141, 1995.

[3] K. Rustan M. Leino, James B. Saxe, Raymie Stata *Checking Java programs via guarded commands*, SRC Technical Note, 1999.

[4] *Boogie*, github.com/boogie-org/boogie

[5] Elmar Keij *Static Testing: Using the Weakest Pre-condition Calculus*, master thesis, Utrecht University Dept. of Information and Computing Sciences, 2009.

[6] Mana Taghdiri *Inferring Specifications to Detect Errors in Code commands*, Automated Software Engineering, pp. 87-121, 2007.

[7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, Chen Xiao *The Daikon system for dynamic detection of likely invariants*, Science of Computer Programming 69.1, pp. 35-45, 2007. plse.cs.washington.edu/daikon

[8] Arno Pol *Clustering and Dynamic Invariant Detection*, master thesis, Utrecht University Dept. of Information and Computing Sciences, 2015.

[9] I.S.W.B. Prasetya, J. Hage, A. Elyasov *Exploiting Annotations to Test Break-off Branches*, 22th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2015.

[10] Jinlin Yang, David Evans *Dynamically Inferring Temporal Properties* , Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. ACM, 2004.

[11] Alexander Elyasov, I.S. Wishnu B. Prasetya, Jurriaan Hage *Guided Algebraic Specification Mining for Failure Simplification*, IFIP International Conference on Testing Software and Systems. Springer Berlin Heidelberg, 2013.

[12] Davide Lorenzoli, Leonardo Mariani, Mauro Pezzè *Inferring State-based Behavior Models*, Proceedings of the 2006 international workshop on Dynamic systems analysis. ACM, 2006.

[13] Carlo Alberto Furia, Bertrand Meyer *Inferring Loop Invariants Using Postconditions*, Fields of Logic and Computation. Springer Berlin Heidelberg, 2010.

[14] K. Rustan M. Leino, Francesco Logozzo *Loop invariants on demand*, APLAS 2005.

[15] Tao Xie, David Notkin *Exploiting Synergy Between Testing and Inferred Partial Specifications*, In WODA, pp. 17-20, 2003.

[16] *Jtest*, ParaSoft Corportation, parasoft.com

[17] H. K. N. Leung, L. White *A cost model to compare regression test strategies*, Proc. Conf. Softw. Maint., pp. 201-208, 1991.

[18] *Haskell*, haskell.org

[19] René Just *The Major mutation framework: Efficient and scalable mutation analysis for Java*, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 433-436, 2014.

[20] I.S.W.B. Prasetya *T3, a Combinator-based Random Testing Tool for Java: Benchmarking*, Int. Workshop Future Internet Testing, Lecture Notes in Computer Science, 8432, Springer, 2014.

[21] Cu D. Nguyen, Alessandro Marchetto, Paolo Tonella *Automated oracles: An empirical study on cost and effectiveness*, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 136-146, ACM, 2013.

[22] L. de Moura, N. Bjørner *Z3: An efficient SMT solver*, In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 08), 2008.

[23] *Apache Commons*, commons.apache.org