Using Symbolic Execution to Guide Test Generation

Gareth Lee†, John Morris†, Kris Parker†, Gary A Bundell† , Peng Lam‡

***Abstract*—**
This paper examines the efficacy of symbolic execution as a method for automatic test pattern generation. We focus on the creation of test patterns for software components written in Java.
A number of shortfalls of symbolic execution have been discussed in previous publications, such as dealing with loops, method calls, impossible paths and algebraic simplification. We present practical solutions to several of these problems which we have implemented in the form of a software component testing system. We also present a novel approach for dealing with aliasing of array elements and references within symbolic execution.

Date: March 13, 2001 at 17:37

I. Introduction

A software component of even moderate complexity has so many paths which can be followed during execution with real data - each of which must be tested if the component is to be labelled 'fully verified' - that automated generation of the necessary tests has attracted research interest for at least two decades. The problem is fundamentally a hard one: if a method of a component has n simple two-way branches (`if .. then .. else` statements), then there may be as many as 2^n paths to be tested.
An automatic test generation system based on a simple parser can parse the code of a method with simple branches, identify the n branches and enumerate the paths which must be followed for a comprehensive set of tests. Each such path represents an equivalence class for testing the component and a representative (set of values of the inputs) of the equivalence class must be chosen and *values of the outputs resulting from applying the method to the inputs determined.* Symbolic execution[1], [2], [3] can produce formal specifications for the equivalence classes in the form of constraints which each input must satisfy. However, it is determining outputs which the chosen representative should produce which is the expensive process. Except in rare cases where formal, transformable specifications have been produced, the specifications are expressed in natural language. Correct outputs can only be determined, from a natural language specification, by a human who can read and interpret its meaning. Thus automatic path detection and test generation systems must rely on the absence of gross errors such as the generation of exceptions to detect faults as there is no practical way to automatically generate outputs in accordance with the specifications.

†Centre for Intelligent Information Processing Systems, Department of Electrical and Electronic Engineering, The University of Western Australia, Nedlands WA 6907, Australia. email: `[gareth,morris,kaypy,bundell]@ee.uwa.edu.au`
‡School of Engineering, Murdoch University, Murdoch WA 6150, Australia. email: `peng@eng.murdoch.edu.au`

Despite this limitation, automatic test pattern generation (ATPG) has a useful role in maintaining quality software:

- It provides a very strict basis for regression testing of reasonably stable components. The stability criterion is important since
  - test cases are only reusable if the interface to a specific method does not change during modification
  - if the implementation of a method changes dramatically, then the (implementation-derived or 'white box') equivalence classes may also change.
- The equivalence classes provide a formal and compact description of the behaviour of the component. Checking them against the original requirements provides very strong support for the integrity of the implementation.

However, researchers who have previously studied symbolic execution have noted some significant problems which limit its ability to define accurately individual equivalence classes[4], [5]:

1. Handling loops within the source code[3],
2. handling onward method calls[6],
3. array element aliasing[6],
4. infeasible paths[3],
5. the cost of algebraic simplification[1],
6. pointer/reference aliasing and
7. handling other complex abstract data types.

Although it is widely accepted that symbolic execution is the strictest way to generate an exhaustive set of test patterns, these difficulties have resulted in relatively little interest in symbolic execution in recent years. Attempts to apply symbolic execution to languages whose use of pointers has been (appropriately) described as 'promiscuous' [2] has led to complex and rather impractical solutions[7].
However, many recent languages tend towards formal data typing, which makes them well suited to analysis by symbolic execution. Java is one example, which uses strictly typed object references as a consequence of its design requirement for a secure sand-box execution model[8]. It is likely that future imperative languages, such as Microsoft's C-sharp[9], will continue the type-safe approach. We believe this trend, combined with substantial increases in processor performance, mark an opportunity to revisit symbolic execution as an automatic test pattern generation (ATPG) technique.

[1]The final three items arise from our own research in this area.
[2]This statement has entered programming language folklore: we are unable to locate the original source, but we believe it is a widely accepted truth.

We have implemented a symbolic execution system for use with software components written in Java. Software components provide the perfect testing ground for symbolic execution since they are significantly smaller than traditional 'monolithic' software systems. This makes them easier to test. Software components are also typically designed for reuse allowing the cost of testing to be amortized over a greater number of users.
Our 'Component Test Bench' (**CTB**)[10] provides an environment for developing solutions to some of the symbolic execution shortcomings listed above. The project objective is to create an ATPG system which can be used to analyse any syntactically valid source code. In the following sections we illustrate the idea behind symbolic execution and how it can be used for ATPG and then discuss the implementation of the **CTB** and propose a solution to the array index aliasing problem.
The terminology used throughout this paper is that of object orientated program development and Java. However, the symbolic execution concepts discussed here are not restricted to OO languages so the reader should feel free to substitute non-OO synonyms: for instance *procedure* or *function* in place of *method*.

II. Symbolic execution

Literal execution requires that the program be executed with actual values that are bound to variables and modified by statements within the program. These values steer the flow of control through the program based on the outcome of a number of conditional tests. This is a model that all imperative programmers understand.
Symbolic execution is a technique for analysing the dataflow dependency along a preordained path through a method. The variables are bound to symbolic (often algebraic) values and their interdependence can be tracked via algebraic expressions which are produced as the program executes. On completion, symbolic execution gives rise to two sets of expressions:

- An expression which describes the set of conditions that must be met by the parameters to ensure that the chosen path will be followed. Commonly called the 'path condition' (PC).
- An set of expressions which fully describe the side effects of executing the method. This is a set of mappings which describe how its output variables depend on its formal parameters.

The path condition for a specific path defines an *equivalence class*. This means that all test patterns that are members of the class will follow the same path testing the method in the same way. In this sense they are all equally able to to detect a fault along the path and are therefore *equivalent*.
Consider the example method shown in figure 1. The `testMethod` has two parameters x and y and returns a single value which we will denote ret [3]. It is comprised of six statements which we denote $S0$ to $S5$.

```
     public static int testMethod(int x, int y)
     {
S0:    if (x > 3) {
S1:       int z = x + 2;
S2:       if (z < y)
S3:          return x;
          else
S4:          return y;
       }
S5:    return 0;
     }
```

Fig. 1. A Java method that could be analysed by symbolic execution. The six statements within the program are labelled $S0$ to $S5$.

Symbolic execution of this method will give rise to three equivalence classes:

- Equivalence class one corresponds to the path $S0$;$S5$. The path condition is $(x \leq 3)$ and the mapping is $ret = 0$.
- Class two corresponds to the path $S0$;$S1$;$S2$;$S3$. The path condition is that $(x > 3)$ and $(x + 2 < y)$ must jointly be satisfied, whilst the mapping is $ret = x + 2$.
- Class three corresponds to the path $S0$;$S1$;$S2$;$S4$. The path condition is that $(x > 3)$ and $(x + 2 \geq y)$ must jointly be satisfied, whilst the mapping is $ret = y$.

In combination the set of equivalence classes fully characterise the behaviour of the method. They can also be used to generate test patterns to exercise the code later during regression testing. This can be achieved by choosing a test pattern for each of the equivalence classes with values which satisfy the associated path condition. Expected results can be generated for each test pattern by sustituting the the chosen parameters into the result mapping functions.
For the three equivalence classes listed above we might generate three test patterns using randomly chosen x and y (shown in figure 2).

Equivalence class	Input x	Input y	Result ret
1	-4	6	0
2	7	15	9
3	9	11	11

Fig. 2. Three test patterns derived from the equivalence classes.

Used in this way symbolic execution provides a very precise and complete mechanism for test pattern generation.

III. The Symbolic Executor

The are two approaches to implementing symbolic execution systems in Java. Execution can occur on the intermediate form of the language (the byte codes)[11] or

[3]When considering a non-static method, any instance variables that the method uses as r-values must be added to the parameter list and any variables that appear as l-values must be added to the list of results. The fact that these methods are instance variables of an object (rather than formal parameters) does not otherwise impact symbolic execution.

on the original expression of the program (the program source). Most Java programs are compiled into an intermediate form for reasons of compactness and performance. However in the process of compilation much of the original semantic structure is obfuscated - particularly when an optimising compiler is used.
We have therefore chosen to implement a source interpreter as the basis for the component test bench (**CTB**). This parses the source code and builds a parse tree representation of the program statements in memory which is then executed directly by an interpreter [4]. This is referred to as the 'instrumented runtime system' (**IRTS**) since it is configured to collect information about the running program. The **IRTS** is a subsystem within the **CTB**.

A. Overview

The **IRTS** has been designed to be as flexible as possible. It supports execution using literal quantities (in which case it behaves identically to any other Java virtual machine). This allows test patterns to be 'walked-through' much as is possible with a symbolic debugger. An extension to the **IRTS** allows the system to operate with symbolic dataflow paths, allowing the creation of equivalence classes from which test patterns may be derived.
The **IRTS** has two phases of operation: we term these the *dataflow/recording phase* and the *constraint/implication phase*.

A.1 The dataflow/recording phase

In the dataflow phase, the system follows the chosen path and keeps track of the algebraic values bound to each of the variables. For example, refering to figure 1, at the conclusion of path $S0;S1;S2;S3$ the **IRTS** would have three program variables within its scope: parameter **x** with final (algebraic) value x, parameter **y** with value y and local variable **z** with final value $x + 2$.

A.2 The constraint/implication phase

The **IRTS** will then enter the constraint phase in which it will attempt to derive the path condition on the basis of parameters to the method. The initial path conditions would be $(x > 3)$ and $(z < y)$ and it will have to substitute a value for the local variable **z** to obtain a path condition defined in terms of parameters **x** and **y**. When the algebriac expressions arise from an iterative path they will often be convoluted (such as $(((((x+2)-2y)+4x)-7)+4y))$ and they must be simplified to a canonical form before thay can be used. If this is the case, the constraint/implication phase must take responsibility for (computationally expensive) algebraic manipulation.

B. Design Issues

In the introduction a number of shortcomings of previous symbolic execution systems were listed. During the design phase of the **IRTS** we were able to address a number of these problems.

B.1 Loop Handling

The handling of loops within methods must be addressed in a heuristic fashion. Any loop within a method-under-test whose termination condition is functionally dependant on one or more of the parameters will have no clear termination condition. In principle, such a loop could be traversed an infinite number of times. In practice a decision must be made as to which paths to follow: for instance the loop should be traversed zero times, once, twice and a larger number of times. However, it is conceivable that such an approach may miss faults that only arise after a considerable number of iterations (such as an array overflow [5]). One by-product of our array element aliasing solution (which is discussed below in section V) is that it provides an additional set of conditions that ensure no array overflows occur. These can be used as additional preconditions on methods.

B.2 Method Calls

There are two approaches that can be followed when a method which is being analysed by symbolic execution makes calls to other method calls: the *macro-expansion approach*[12] and the *lemma approach*[13], [14]. Both of these approaches are used by the **IRTS** depending on the context of the call.
Symbolic execution is normally used to analyse methods which are part of the public interface to a component. When these public methods make calls to private methods within the same component they are most appropriately made using macro-expansion. This means that the code is 'in-lined' within the interpreter (and the parameters appropriately initialised from those of the caller). This increases the number of potential paths to the product of the feasible paths through caller and callee methods.
However, when public methods are called, the lemma approach may be used since this increases the number of equivalence classes directly without the cost of considering more paths. The lemma approach requires that the equivalence classes for the called method have already been established. This increases the number of equivalence classes to the product of those associated with the callee and caller methods.

B.3 Impossible Paths

Many of the multitude of potential paths that may be followed through a method-under-test, will be impossible to actually follow. This is due to the logical conditions that predicate the various statement blocks. Consider the code snippet in figure 3. The paths $S0;S1;S2;S3$ and $S5;S6$ are possible for values of x greater than 6 and less than or equal to 3 respectively. However, the path condition for

[4]One (minor) weakness with this approach is that it assumes the compiler's code generation algorithm is correct, since we are unable to test this part of the development process.

[5]In Java such faults result in program termination, since an exception is thrown, and so it is highly unlikely that such faults would go undiscovered in a shipped software component.

```
S0: if (x > 3)
S1:    something;
S2: if (x > 6)
S3:    something else;
S4: if (x <= 3)
S5:    final thing;
```

Fig. 3. An code snippet with impossible paths. The six statements are labelled $S0$ to $S5$.

the path $S0$;$S1$;$S2$;$S3$;$S5$;$S6$ would require $(x > 6)$ **and** $(x \leq 3)$ which is impossible, assuming statement blocks $S1$ and $S3$ do not alter the value of x.

The **IRTS** addresses this problem by using a rule-based solution engine which searches for logical impossibilities each time a condition is appended to the path condition during the dataflow/recording phase. As soon as it detects a logical inconsistancy the analysis of the path is aborted. Thus the **IRTS** does not waste resources investigating impossible paths.

B.4 Algebraic Simplification

As discussed in section III-A, the **IRTS** needs to perform algebraic simplification as part of its constraint/implication phase. The simplification subsystem used by the **IRTS** has two aims:

- It converts each expression into a canonical form by removing any redundancy. For instance it would convert the expression $(((((x+2)-2y)+4x)-7)+4y)$ into the equivalent form $((5x+2y)-5)$. This aids human readability in the reports that are created.
- The simplification subsystem acts as a front-end for other algebraic systems within the **IRTS** such as the solution engine used by the array element alising detector (discussed in section V). The canonicalised input format considerably reduces the complexity of these systems.

Our simplifier implementation consists of a database of 186 rules. Each rule consists of two parts: a pattern that must be matched before the rule is enacted and a rewrite form that can replace the matching part of the expression. For instance, the database contains a rule of the form $(N_1E_1) + (N_2E_1) \mapsto (N_1 + N_2)E_1$. We have a simple (but fast) pattern matcher that searches for matches between an expression to be simplified and the patterns associated with each of the rules. It will match N_1 and N_2 in the rule against any numeric literals, but will only match E_1 against a repeated algebraic expression (which must be identical in both instances). For example, the expression $3(z + 5) + 5(z + 5)$ would match the rule and would be rewritten as $(3 + 5)(z + 5)$. This expression would subsequently match with a second rule telling the simplifier that it can evaluate the sum of two numeric literals. Then a third rule would then match, mapping $8(z+5)$ to a sum of products form. The final canonical form of the expression would therefore be $(8z + 40)$ since, at this point, no more rules match.

Since our simpilifier design emphasises simplicity and speed, we decided to progressively simplify expressions as they are constructed during the dataflow/recording phase. This means that the complexity of expressions is kept under control at all times and there is less chance of encountering an expression exceeding the capabilities of the simplifier. Moreover, by keeping most of the complexity of our simplifier in the pattern database it allows us to extend the set of rules as future needs demand.

B.5 Other Problems

We also address several of the other problems listed in the introduction. The most substantial of these is the handling of array element aliasing, described in section V below. The related problem of pointer alising arises when a Java method operates on two parameters which reference the same underlying object. A modification of the mutable object using either reference leads to an apparent change of the other value 'behind-the-scenes'. It is therefore essential that the symbolic execution system be able to detect the implications of such aliases. This is discussed in section V-D below.

We are also currently investigating the problem of supporting symbolic execution of non-numeric ADTs. This is described further in section VI below.

IV. Analysis of a Sample Method

To demonstrate the capabilities of the **CTB** system we have symbolically analysed an implementation of the quicksort partition algorithm[15]. The source code for this test is included in figure 4. This method partitions a subsection of an array of integers a into two groups: only elements between indices *low* and *high* (inclusive) are processed. The first group is comprised of those elements less than the pivot value, whereas the second group is all other elements. To simplify the discussion, the method excludes the recursive method calls that would be required to form a complete sort [6].

A. Path Generation

The **IRTS** contains a path generation algorithm which is used to enumerate potential paths through the method's source code. The symbolic executor subsequently analyses these paths to decide which paths are possible. Firstly the path generator analyses the source representation of the method to detect the loop and control constructs and their relationships. It constructs a heirarchical tree of the these nodes and their nesting. In the case of figure 4 this would result in $C0$ being the root node with three siblings $C1$, $C2$ and $C3$.

Loops are processed in a heuristic manner. The path generator is therefore configured with the strategies that it will apply to various types of control construct:

[6]It is possible for symbolic execution to be used to analyse recursive methods but that is beyond the scope of this article. All recursive methods may be re-expressed in an iterative form, on which symbolic execution is able to operate.

```
   public static
   int partition(int[] a, int low, int high)
   {
      int pivot_item = a[low];
      int left = low + 1;
      int right = high;

C0:   while ( left < right ) {
C1:      while( a[left] < pivot_item )
            left++;

C2:      while( a[right] > pivot_item )
            right--;

C3:      if ( left < right ) {
            int temp = a[left];
            a[left] = a[right];
            a[right] = temp;
         }
      }

      a[low] = a[right];
      a[right] = pivot_item;
      return right;
   }
```

Fig. 4. The quicksort partition method. The four conditional statements within the method are labelled $C0$ to $C3$.

- For each `if` statement block the path generator will create two paths: one that passes through the statement block predicated by `if`, the other through the 'else' clause (if it exists).
- For each `while` loop the path generator will create $(L+1)$ paths that traverse the loop $0..L$ times. `for` loops are treated identically to `while` loops.
- Each `do` loop is treated similarly to `while` except that it is impossible to traverse a loop zero times (since the termination condition is not tested until the end). Since they must be traversed at least once, L paths with $1..L$ iterations are created.
- For each `switch` statement block, it creates paths that visit each of the `case` options (and the `default` option if it is specified). For a switch block with NC case options and a default specified, it will therefore create $NC+1$ paths.

The limit value L can be specified as a parameter to the path generator subsystem. The path generator is designed to enumerate each possible *combination* of paths through the control statements hence, as L increases, the number of potential paths grows very rapidly. The number of potential paths for the `partition` method for L is shown in figure 5. An algebraic expression for the number of potential paths is derived in Appendix A.

The path generator returns each path encoded as a string of true and false values. For instance the path descriptor 'T TTF TF F F' tells the symbolic executor to (execute

Loop limit, L	Potential Paths, $P(C0)$
1	9
2	343
3	33,825
4	6,377,551
5	1,961,791,488

Fig. 5. The number of potential paths for chosen loop limit value L.

the three leading statements, then) enter loop $C0$, traverse loop $C1$ twice but exit the loop on the third test, traverse loop $C2$ one time, do not enter the `if` clause $C3$ (the `else` block would be executed if it existed). The final false value instructs the symbolic executor to exit the `while` loop $C0$, at which point the three trailing statements are executed and the test completes.

B. *Analysis of the* `partition` *method*

We ran the symbolic executor across each of the 343 potential paths resulting from using limit value, $L = 2$. Of the 343 paths only 97 (or 28%) were possible given the conditions imposed on the loop statements. The test ran in 226 seconds on a Celeron 350MHz PC.

The test left us with a 97 sets of path conditions and the functional mapping associated with each path. In this case the (rather artificial) return value is not of much interest since it merely indicates how many times the `right` variable has descended down the array (as a result of iterations of loop $C2$).

For instance, potential path 224 (of the original 343) was described by 'TTFTTFTTTFFFF' and proved to be possible. The path conditions associated with this test are $(high > (low+4)) \wedge (a[low+1] < a[low]) \wedge (a[low+2] = a[low]) \wedge (a[high] > a[low]) \wedge (a[high-1] > a[low]) \wedge (a[high-2] < a[low]) \wedge (a[low+3] \geq a[low]) \wedge (high \leq (low+5))$ [7]. These eight conditions must be jointly satisfied if the path is to be followed.

The only side effect of this static method is its return value which takes the form `(symbolic int (high-2))`. Note that, even when executing symbolically, the **IRTS** is able to track the types of values that are created. The value $(high-2)$ is simply a reflection that the path only traverses loop $C2$ twice during the first iteration of $C0$ (and zero times during the second iteration of $C0$).

It is also enlightening to study a path that proved to be impossible. We will consider potential path 15 associated with path descriptor 'TTTFFTF'. The path conditions at the point that the path terminated were $(high > (low+3)) \wedge (a[low+1] < a[low]) \wedge (a[low+2] < a[low]) \wedge (a[low+3] \geq a[low]) \wedge (a[high] \leq a[low]) \wedge (high \leq (low+3))$. Just prior to termination the symbolic executor had entered the `if`

[7]The rather awkward form of this expression illustrates why this process is best automated. Moreover, the expression demonstrates that we can take further steps in the algebraic simplification of information we present to the user. Combining the conditions $high > (low+4)$ and $high \leq (low+5)$ along with knowledge that low and $high$ are of type integer, would allow us to conclude more succinctly that $high = (low+5)$.

```
public static
int rotate(int[] array, int i, int j, int k)
{
   int temp = array[i];
   array[i] = array[j];
   array[j] = array[k];
   array[k] = temp;
   return array[i];
}
```

Fig. 6. A method which rotates the contents of three array elements.

statement block $C3$ which requires that $((low+3) < high)$. The statement block of $C3$ does not modify the values of variables $left$ or $right$, so when the path immediately tried to exit loop $C0$, this could only happen if $((low+3) \geq high)$. The **IRTS** immediately detected this contradiction and terminated execution of the path.

V. Array Element Aliasing

Consider the method listed in figure 6. This is a very simple method that rotates the contents of three elements contained in an array of integers (formal parameter *array*).
An experienced programmer would probably attach a set of preconditions to this method. For instance *array* must already be allocated, parameters i, j and k must not exceed the number of elements within *array* and finally $i \neq j$, $j \neq k$ and $i \neq k$.
These final three preconditions are the most interesting. Sometimes it is not possible to know whether these conditions have been met by the caller. If any two of these values coincide they profoundly effect the execution behaviour of the `rotate` method since storage of the three array elements will no longer be distinct. This is the *array element aliasing problem.*
It is clear that accesses to $array[i]$ and $array[j]$ will refer to different storage locations when $i \neq j$. But when $i = j$ they will refer to the same storage location and so sucessive accesses as l-values will return the same value. Moreover, accesses as r-values may result in the shared storage location being overwritten.

A. The Element Aliasing Subsystem

Our extension to the symbolic executor locates situations where array acesses may result in the aliasing of the underlying storage locations. It ascertains how many distinct categories of aliasing may occur and it generates a set of preconditions that, if satisfied, will ensure that a specific category of aliasing results or conversely that no aliasing is possible.
We refer to each category of aliasing as an *execution context* (EC). Once it has decided on the possible ECs, the symbolic executor will execute each context again to see if the context alters the functional mapping for the method. As we will see in the case of `rotate` the return value depends on the EC which is used.
Our subsystem starts by performing conventional symbolic execution on a method and detecting instances where different algebraic indices are used to access elements within the same array. If it found two access such as $array[i+2j]$ and $array[5j]$ it would infer algebraically that, if the *unification condition* ($i = 3j$) is imposed, both accesses will refer to the same storage location. It progresses to work out every possible combination of unifications and generate an EC for each combination (including the conditions and EC when every array access is distinct).

B. The `rotate` example

To illustrate the possible ECs, we will examine the code sample in figure 6. The element aliasing subsystem is able to resolve five distinct ECs for this method (only one of which the author presumably intended). The side effects of the ECs are shown in figure 7. In this table we use a notation to describe the storage locations that the program uses during execution. If the storage location behind $array[i]$ is distinct from all others we denote this s_i in the table. However, if some unification condition is in force which means that $array[j]$ and $array[k]$ are storing to the same location we denote this s_{jk}. In EC_5 all array indices are equal and so all accesses are to the same storage location, denoted s_{ijk}.
The table provides the unification conditions for each EC. It also shows the sequence of storage operations that the method will perform. It demonstrates that many of the operations become redundant in the presence of unifications. Indeed EC_5 devotes considerable effort to reading and rewriting the same storage element five times with no side effect at all. Finally the table lists the parameter value the method returns. This will be value of one of the elements within *array* before the method was called. In some cases the unification conditions imply that two or more of the original array elements were unified from the outset and so the possible return values are listed. It is interesting to observe that in each case the side effects of the method are different since it returns a different value from within the original *array*.

C. Element Aliasing Implementation

Our subsystem automates the process that was just described by adding an additional phase after the constraint/implication phase. Once the path conditions have been derived, it further scans the method to detect array indices and construct a set of unifications constraints and hence an execution context. These may be thought of as adjunct to the PCs since they do not indicate which path is followed, but rather how the execution within that path behaves.
The symbolic executor is then restarted for each of the ECs. It performs the dataflow analysis as shown in figure 7 allowing the correct side effects for the EC to be ascertained. Used for ATPG this would allow a test pattern to be chosen satisfying each EC, rather than just one for the entire PC as was previously possible. If the tester wishes to rule out one of more unexpected ECs they have the option of im-

Exec. Context	EC_1	EC_2	EC_3	EC_4	EC_5
Unification Conditions	$i \neq j$ $i \neq k$ $j \neq k$	$i = j$ $i \neq k$ $j \neq k$	$i \neq j$ $i \neq k$ $j = k$	$i \neq j$ $i = k$ $j \neq k$	$i = j$ $i = k$ $j = k$
Storage Operations	$s_i \rightarrow s_{temp}$ $s_j \rightarrow s_i$ $s_k \rightarrow s_j$ $s_{temp} \rightarrow s_k$ $s_i \rightarrow return$	$s_{ij} \rightarrow s_{temp}$ $s_{ij} \rightarrow s_{ij}$ $s_k \rightarrow s_{ij}$ $s_{temp} \rightarrow s_k$ $s_{ij} \rightarrow return$	$s_i \rightarrow s_{temp}$ $s_{jk} \rightarrow s_i$ $s_{jk} \rightarrow s_{jk}$ $s_{temp} \rightarrow s_{jk}$ $s_i \rightarrow return$	$s_{ik} \rightarrow s_{temp}$ $s_j \rightarrow s_{ik}$ $s_{ik} \rightarrow s_j$ $s_{temp} \rightarrow s_{ik}$ $s_{ik} \rightarrow return$	$s_{ijk} \rightarrow s_{temp}$ $s_{ijk} \rightarrow s_{ijk}$ $s_{ijk} \rightarrow s_{ijk}$ $s_{temp} \rightarrow s_{ijk}$ $s_{ijk} \rightarrow return$
Return Value	$array[j]$	$array[k]$	$array[j]$, $array[k]$	$array[i]$, $array[k]$	$array[i]$, $array[j]$, $array[k]$

Fig. 7. The various execution contexts arising from the `rotate` method.

posing preconditions on the method-under-test to restrict its use.

We further analysed the quicksort partition example (previously discussed in section IV-B) to see what execution contexts arose. Within the 97 feasible paths our system detected 232 possible execution contexts. It increased the number of separate tests which should be applied to the method by 140%.

D. Reference Aliasing

A related form of aliasing can arise with the parameters of the method-under-test. Reference aliasing will impact the operation of symbolic execution when the following conditions are met:

- Two or more parameters to the method-under-test are type compatible.
- Their class (or interface) type must be mutable.
- Their references are used within the chosen path of the method-under-test to modify the state of the objects by calling one or more *setter* methods.

A simple Java example which meets these conditions is shown in figure 8, which uses references to the `java.util.Set` type.

```
public static Set aliasable(Set x, Set y)
{
   Object three = new Integer(3);
   x.remove(three);
   if (y.contains(three))
      return x;
   return null;
}
```

Fig. 8. A method which is susceptible to reference aliasing.

Assume the chosen path requires that the `if` block be entered. Two ECs exist for this path. The first EC will result from the unification condition ($x \neq y$). This is presumably the expected mode of operation since x and y are distinct. The path condition will be ($3 \in y$).

A second EC will arise when ($x = y$) (since both references refer to the same set instance), in which case invocation of the method will behave very differently. Now the call to `x.remove()` will remove the element 3 from the shared set and so it becomes impossible to follow the chosen path, since `y.contains(three)` cannot subsequently be true.

Our **IRTS** implementation does not currently deal with this form of aliasing. We will add this capability as part of our future development work.

VI. Ongoing work

We are currently looking at the problem of symbolic execution of abstract data types. These are values that can propogate within the program but do not follow algebraic rules. The most widely used example is Java's `java.lang.String` type, but we could also consider types such as `java.util.Set`. There are two approaches that we plan to investigate.

Firstly one could inline them, decomposing the `String` for instance, into an array of character ordinals and a length integer. Sets of equivalance classes could then be constructed for each of the public interfaces of `String`. Subsequently, when `String` is cited in any tests, it could be treated as a sequence of method calls to a separate software component, using the lemma approach (see section III-B.2).

The second approach is based on the observation that, although this solution would work, it does not provide the most compact and useful description of what is happening to each `String`'s internal state as it propogates through the program. Strings are subject to operations within the program which are not algebraic, such as concatenation and conversion to upper case. It might therefore be better to model them with some symbolic value which is better suited to describing the contents of a `String`. Considering the objective of symbolic execution is to allow sets of test patterns to be created, a better representation for the contents of a `String` might be a regular expression.

We will propose an extension to symbolic execution to deal with these types in a future publication.

VII. Conclusion

This work demonstrates the efficacy of symbolic execution as a means of automatic test pattern generation. We believe that the approach is particularly applicable to the semi-automated testing of software components, due to their limited size. Widespread reuse of components also allows the cost of testing to be amortized over a large user base.

Our research aims to generalise symbolic execution to a point where it can be applied to any Java source code. Future work may allow symbolic execution to be used with other strongly typed languages.

This work on symbolic execution has been part of a unified approach towards software component testing and marketing, which includes the development of a standard XML document format for the interchange of test pattern sets, such as those created by the **CTB**[10].

The project also operates a software component trading web site, VeriLib[16]. We encourage our contributors to test their software components using the **CTB** and the symbolic execution techniques that have been described here. We believe components that have been rigorously tested will be at a commercial premium over coming years. For further information regarding the future development of the **CTB** we encourage readers to periodically visit the VeriLib site.

VIII. Acknowledgments

This work was supported by a grant from Software Engineering Australia (Western Australia) Ltd through the Software Engineering Quality Centres Program of the Department of Communications, Information Technology and the Arts.

References

[1] James C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul 1976.

[2] L. A. Clarke, "Symbolic execution as an aid to testing and verification," in *Software Verification and Validation.* Darmstadt, RFA, Sep 83.

[3] P. D. Coward, "Symbolic execution and testing," *Jnl Information and Software Technology*, vol. 33, no. 1, pp. 53–64, Feb 1991.

[4] Glenford J. Myers, *The Art of Software Testing*, Wiley - Interscience, New York, 1979.

[5] I. Sommerville, *Software Engineering*, Addison-Wesley, London, 1982.

[6] P. David Coward, "Symbolic execution systems – a review," *Software Engineering Journal*, pp. 229–239, Nov 1988.

[7] Berhard Scholz, Johann Blieberger, and Thomas Fahringer, "Symbolic pointer analysis for detecting memory leaks," *ACM SIGPLAN Notices*, vol. 34, no. 11, pp. 104–113, Nov 1999.

[8] Li Gong, "Java security architecture (JDK 1.2)," Tech. Rep., JavaSoft, Jul 1997, `http://www.javasoft.com/products/jdk/preview/docs/guide/security/index.%html`.

[9] Microsoft Corporation, *C# Introduction, Overview and Language Specification*, `http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp`, 2000.

[10] Gary A Bundell, Gareth Lee, John Morris, and Peng Lam, "A software component verification tool," in *Proceedings: International Conference on Software Methods and Tools.* Nov 2000, IEEE Computer Society Press / ACM Press.

[11] Parasoft Corporation, Monrovia, Texas, USA, "Method and system for generating a computer program test suite using dynamic symbolic execution of java programs," U.S. Patent No. 5,784,553, Apr 1997.

[12] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT – A formal system for testing and debugging programs by symbolic execution," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 234–245, Jun 1975.

[13] Sidney L. Hantler and James C. King, "An introduction to proving the correctness of programs," *ACM Computing Surveys*, vol. 8, no. 3, pp. 331–353, Sep 1976.

[14] Thomas E. Cheatham, Jr., Glenn H. Holloway, and Judy A. Townley, "Symbolic evaluation and the analysis of programs," *IEEE Transactions on Software Engineering*, vol. 5, no. 4, pp. 402–417, Jul 1979.

[15] C. A. R. Hoare, "Quicksort," *Computer Journal*, vol. 5, no. 1, pp. 10–15, 1962.

[16] Software Component Laboratory, *VeriLib: A Source of Reliable Components*, `http://www.verilib.sea.net.au`, 2000.

Appendix

I. Calculating the multiplicity of paths

The number of paths associated with any tree of control nodes can be specified by the recurrence relationship.

$$P(N_i) = \sum_{j=M_i}^{S_i} \left[\prod_{k=1}^{C_j^i} P(N_{j,k}^i) \right]^{R_j^i} \qquad (1)$$

This describes the number of potential paths through a control node N_i which we denote $P(N_i)$. This is a function of the number of paths through each of the child control nodes, denoted $P(N_{j,k}^i)$, where $N_{j,k}^i$ is the k-th child of N_i when it is in state j. The child control nodes are those control nodes nested within the statement block(s) of N_i. This assumes that node N_i has C_j^i children when it is in state j: for instance, an `if` statement may have one set of children when it visits its `if` statement block (for which $j = 1$) and a different set (and number) of children in its `else` block (when $j = 0$). This also applies to `switch` statements, but not to any of the loop nodes, for which C_j^i and $N_{j,k}^i$ are the same for all j.

The additional parameters M_i and S_i denote the minimum and maximum state for N_i. The interpretation of the state parameter depends on the type of node under consideration:

- For `if` nodes the state selects which of the two possible child blocks (`if` or `else`) is being visited and so C_j^i depends on the choice of j.
- `switch` nodes also use the state as an index for the (`case` or `default`) child block which is being considered.
- In the case of all loops, the state j indicates which iteration of the loop is currently under consideration.

Finally R_j^i denotes the number of times that execution of a path will visit the children as a function of the state. Given the previous discussion we can see that `if` and `switch` nodes visit their children once no matter what their state is. Loop nodes will have visited their children once for each iteration prior to and including j and so $R_j^i = j$.

This recurrence relationship only applies to parent nodes. Table 9 shows trivially the number of paths through childless nodes, denoted $P(N_i)'$, and is equal to $S_i - M_i + 1$. L denotes the limit we have chosen for loop travsersal (see the main text). The table also provides values of the parameters for common control structure types. NC_i is the number of clauses within nodes which correspond to `switch` statements: it counts the number of `case` clauses, plus one if the statement has a `default` clause.

Control type of N_i	$P(N_i)'$	M_i	S_i	R_j^i
`if` statement	2	0	1	1
`while` and `for` loops	$L+1$	0	L	j
`do` loop	L	1	L	j
`switch` statement	NC_i	1	NC_i	1

Fig. 9. Values for parameters $P(N_i)'$, M_i, S_i and R_j^i for various control node types.

We can apply this framework to the `partition` method described in section IV. The method has four control nodes, labelled $C0$ to $C3$ in figure 4. Of these only $C0$, the outer `while` loop, is a parent node to which the recurrence relationship applies. Control nodes $C1$ to $C3$ are children of $C0$, but are themselves childless and so their number of paths, $P(N_i)'$, can be looked up in the table above. We therefore end up with a relatively simple expression for $P(C0)$ when we substitute for the product of the three children.

$$P(C0) = \sum_{j=M_i}^{S_i} [P(C1)'P(C2)'P(C3)']^{R_j^i} \qquad (2)$$

By substituting values from the lookup table for the number of paths through $C1$ and $C2$ (which are both `while` loops) and through $C3$ (which is an `if` statement), we get an expression of the form:

$$P(C0) = \sum_{j=M_i}^{S_i} [(L+1)(L+1)2]^{R_j^i} \qquad (3)$$

simplifying and looking up M_i, S_i and R_j^i for $C0$ (a `while` loop) gives the final form:

$$P(C0) = \sum_{j=0}^{L} \left[2L^2 + 4L + 2\right]^j \qquad (4)$$

This final form gives us a relation between the maximum number of loop iterations that we are prepared to make, L, and the number of potential paths $P(CO)$ through $C0$ that we must investigate. We can see that this function of L is $O(n^{2n})$. Substituting values for L will give rise to the values in figure 5 of the main text.