Test Data Generation for Programming Exercises with Symbolic Execution in Java PathFinder

Petri Ihantola
Helsinki University of Technology
P.O. Box 5400, 02015 TKK
Finland
petri@cs.hut.fi

ABSTRACT

Automatic assessment of programming exercises is typically based on testing approach. Most automatic assessment frameworks execute tests and evaluate test results automatically, but the test data generation is not automated. No matter that such test data generation techniques and tools are available.

We have researched how the Java PathFinder software model checker can be adopted to the specific needs of test data generation in automatic assessment. Practical problems considered are: How to derive test data directly from students' programs (*i.e.* without annotation) and how to visualize and how to abstract test data automatically for students? Interesting outcomes of our research are that with minor refinements *generalized symbolic execution with lazy initialization* (a test data generation algorithm implemented in PathFinder) can be used to construct test data directly from students' programs without annotation, and that intermediate results of the same algorithm can be used to provide novel visualizations of the test data.

Keywords

test data generation, symbolic execution, automatic assessment

1. INTRODUCTION

Besides software industry applications, typical examples where automated verification techniques are applied are numerous assessment systems widely used in computer science (CS) education (*e.g.* ACE [18] and TRAKLA2 [14]) – especially in systems used for automatic assessment of programming exercises (*e.g.* ASSYST [10], Ceilidh [4], and SchemeRobo [17]). Automatic assessment of programming exercises is typically based on testing approach and seldom on deducting the functional behavior directly from the source code (such as static analysis in [19]). In addition, it is possible to verify features that are not directly related to the functionality. For example, a system called Style++ [1] evaluates the programming style of students' C++ programs. The focus of this work is on testing, not on formal assessment. Therefore, the term *automatic assessment* is later on used for test driven assessment of programming exercises.

Testing is used to increase trust about the correctness of a program by executing it with different inputs. Thus, the first thing to do is to select a representative set of inputs. The input for a single test is called *test data*. After the test data has been selected, the correctness of the behavior of the program is evaluated. The functionality that decides whether the behavior is correct or not is called a *test oracle*. The test data and the corresponding oracle together are called a *test case*. Finally, the test data of a logical group of tests together are called a *test set*.

Test data generation can be extremely labor intensive. Therefore, automated methods for the process have been studied for decades (*e.g.* [6]). However, for some reason such systems are seldom used in automatic assessment.

In this work we will apply an automatic test data generation tool, namely Java PathFinder (JPF) [21], in test data generation for automatic assessment. In addition to previously reported techniques of using JPF in test data generation, we will improve these techniques further on. We will also explain how to automatically provide abstract visualizations from automatically generated tests.

Although we discuss automatic assessment and feedback, the core of this research is on automatic test data generation and visualization. We will introduce a new technology we hope to be useful in programming education. However, we are not yet interested in evaluating the educational impact of this work. Manual test data generation is already the dominant assessment approach in programming education. This work makes test data generation easier for a teacher and provides visualizations and better test adequacy for students. Thus, we believe that results of this work are valuable as is.

The rest of this article is organized as follows: Section 2 is about the previous research of others and heavily based on previous work of Willem Visser, Corina Păsăreanu, Sarfraz Khurshid, and others [2, 5, 12, 16, 20, 21]. Section 3 describes our contribution and changes to the previous techniques. Section 4 discusses about some quality aspects in different test data generation techniques and Section 5, finally, concludes the work.

2. AUTOMATIC TEST DATA GENERATION

2.1 Different Approaches

There are several different techniques for automated test input generation. There are also many test generation tools for programs manipulating references introduced in the literature (*e.g.* [3, 11, 22]). Unfortunately most test input generation tools are either commercial or unavailable for some other reason. In addition, many open source testing tools[1] concentrate on other aspects of testing than test input generation. Here we will not describe tools, but some techniques for test data generation. Later in Section 2.2, we will explain how the techniques can be implemented in JPF.

[1] `http://opensourcetesting.org/` [April 10, 2006]

2.1.1 Method Sequences vs. State Exploration

In unit testing of Java programs, test input consists of two parts: 1) explicit arguments for the method and 2) current state of the object (*i.e.* implicit `this` pointer given as an argument). The first decision in test input generation is to decide how object states are constructed and presented. There are at least two approaches to the task:

Method sequence exploration is based on the fact that all legal inputs are results from a sequence of method calls. A test input is represented as a method sequence (beginning from a constructor call) leading to the state representing test data.

Direct state exploration tries to enumerate different (legal) input structures directly (*i.e.* without using the methods of the class in the state construction). Heuristics can also be applied or the state enumeration can be derived from the control flow of the method to be tested (as in Section 2.2.3).

The common justification for using method sequence exploration is that in assessment frameworks, object states can only be constructed through sequential method calls. On the other hand, in the method sequence exploration, tests are no longer testing only a single method. If the methods needed in the state construction are buggy, it is difficult to test other methods. However, in automatic assessment, one might want to give feedback from all the methods of the class at the same time – not to say that feedback from method X cannot be given before problems in method Y are solved.

2.1.2 Symbolic Execution

The main idea behind *symbolic execution* [13] is to use symbolic values and variable substitution instead of *real execution* and real values (*e.g.* integers). In symbolic execution, return values and values of variables of programs are symbolic expressions consisting of symbolic input. For example, the output for a program like "`int sum(int x, int y) { return x+y; }`" with symbolic input `a` and `b` would be `a + b`.

A state in symbolic execution consists of (symbolic) values of program variables, a path condition and the program counter (*i.e.* information where the execution is in the program). Path condition is a boolean formula (or the corresponding constraint satisfaction problem (CSP)) over input variables and describes which conditions must be true in the state. A *symbolic execution tree* can be used to characterize all execution paths (*i.e.* state chains). Moreover, a finite symbolic execution tree can represent an infinite number of real executions. Formally, a symbolic execution tree SYM($\mathcal{P}$) of a program $\mathcal{P}$, is a (possibly infinite) tree where nodes are symbolic states of the program and arcs are possible state transitions.

For example, the symbolic execution tree of Program 1, `min( X, Y )`, is illustrated in Figure 1. In the initial state, input variables have the values specified by the (symbolic) method call and the path condition is *true*. Nodes with an unsatisfiable path condition are pruned from the tree (labeled "backtrack" in the figure).

All the leaf nodes of a symbolic execution tree where the path condition is satisfiable represent different execution paths. Moreover, all feasible execution paths of $\mathcal{P}$ are represented in SYM($\mathcal{P}$). In the example of Figure 1, there are two satisfiable leafs and therefore exactly two different execution paths in Program 1. All satisfiable valuations for a path condition of a single leaf node in SYM($\mathcal{P}$) will give us inputs with identical execution paths in the program ($\mathcal{P}$). Furthermore, all leaf nodes represent different execution paths. Thus, if SYM($\mathcal{P}$) is finite we can easily generate inputs for all possible execution paths in ($\mathcal{P}$) and if SYM($\mathcal{P}$) is infinite the maximal path coverage is unreachable.

The golden age of symbolic execution goes back to 70's. The original idea was not developed for the test data generation, but formal verification and enhancement of program understanding through symbolic debugging. However, the approach had many problems [7] including: 1) Symbolic expressions quickly turn complex; 2) Handling complex data structures is difficult; 3) Loops dependent on input variables are difficult to handle.

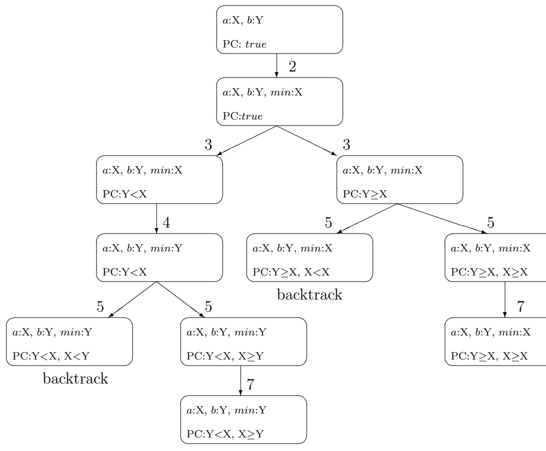

Figure 1: Symbolic execution tree of the Program 1. Numbers in the figure are line numbers.

```
1  int min( int a, int b ) {
2    int min = a;
3    if ( b < min )
4      min = b;
5    if ( a < min )
6      min = a;
7    return min;
8  }
```

Program 1: A program calculating minimum of two arguments. Line 6 is dead code (*i.e.* never executed) as one can see from Figure 1.

2.2 Test Data Generation with JPF

JPF is an open source explicit-state model checker of Java programs. Under the hood it is a tailored virtual machine, and therefore any compiled Java program (*i.e.* byte-code) can be directly used as an input for it. No source-to-source translation is needed as with many other model checkers.

In addition to standard Java libraries, JPF provides some library classes to control the model checking directly from the verified program. The following methods of the Verify class will be applied later in different test data generation strategies:

random(int n) will nondeterministically return an integer from $\{0, 1, \ldots n\}$.

randomBoolean() will nondeterministically return `true` or `false` nopagebreak

ignoreIf(boolean b) will cause the model checker to backtrack if *b* evaluates to true. The method is typically used to prune some execution branches away.

The fundamental idea behind nondeterministic functions is that whenever they are model checked, all the possible values are tried one by one.

JPF provides also a symbolic execution library. The library is not yet publicly available but an evaluation version was obtained for our study. The library provides types like SymbolicInteger, SymbolicBoolean and SymbolicArray. The main idea with the library is to provide model level abstractions for programmers. For example, integer variables are replaced with SymbolicIntegers and operators between integers with methods of the SymbolicInteger class

The symbolic library of JPF keeps track of the path condition. Whenever branching depending on a symbolic variable occurs (*i.e.* some of the comparison methods are called), the execution nondeterministically splits into two, and the condition (or its negation on the else branch) is added to the path condition. The framework uses a standard CSP solver for two tasks:

- Whenever a new constraint is added to the path condition, satisfiability is checked. If the path condition is unsatisfiable, `Verify.ignoreIf(true)` is called and the corresponding execution branch is pruned as the JPF backtracks.
- To provide concrete valuations for (symbolic) input states (*i.e.* to get concrete test data from a symbolic state)

2.2.1 *Explicit Method Sequence Exploration*

Explicit method sequence exploration is based on generating method sequences of different length by using the nondeterministic functions of JPF as in Program 2. The example is a container where states are constructed with insert and delete methods. The example generates all the method sequences up to 10 calls with arguments varying between 0 and 5. Actually, all the possible states of a traditional binary search tree can be constructed by repeating the insert method only, but all the states of the class are not necessarily reached with the same approach. For example, if a binary search tree uses lazy delete, all the states cannot be reached through inserts only.

2.2.2 *Symbolic Method Sequence Exploration*

Symbolic method sequence exploration is similar to explicit method sequence exploration. The only difference is that symbolic variables are used instead of concrete ones. Program 3 does the same as Program 2, but with symbolic values. Because arguments given for the BinarySearchTree are no longer integers but symbolic integers, the original container class needs to be annotated before the symbolic approach can be used. The annotation means that integers are replaced with SymbolicIntegers and operators with the corresponding method calls.

2.2.3 *Generalized Symbolic Execution with Lazy Initialization*

Generalized symbolic execution with lazy initialization, described by Visser *et al.* [12, 21] is a symbolic state exploration technique. In contrast to method sequence exploration, the approach does not require a priori bounds of the input structures (*e.g.* `END_CRITERIA` in Programs 2 and 3). Ideally the approach uses only the method to be tested in the test data generation. Thus, test data can also be generated to methods in a partially implemented class with some relevant methods missing.

```
public static final int END_CRITERIA = 10;
public static final int MAX_ARGUMENT = 5;
public static void main(String[] args) {
  Container c = new BinarySearchTree();
  for ( int i = 0; i <= END_CRITERIA; i++ ) {
    if ( Verify.randomBoolean() ) break;
    if ( Verify.randomBoolean() )
      c.delete( Verify.random(MAX_ARGUMENT) );
    else
      c.insert( Verify.random(MAX_ARGUMENT) );
  }
}
```

Program 2: Test data creation with explicit method sequence exploration for a `BinarySearchTree` class.

```
public static final int END_CRITERIA = 10;
private static void main(String[] args) {
  Container c = new BinarySearchTree();
  for ( int i = 0; i <= END_CRITERIA; i++ ) {
    if ( Verify.randomBoolean() ) break;
    if ( Verify.randomBoolean() )
      c.delete( new SymbolicInteger() );
    else
      c.insert( new SymbolicInteger() );
  }
}
```

Program 3: Test data creation with symbolic method sequence exploration for the annotated `BinarySearchTree` class.

The program to be tested is annotated so that fields are lazily initialized when they are first used. Special getters and setters have to be written for each field of the class. After that, fields are used through these methods only. When an unused (no previous reads or writes) field of a reference type is accessed through a getter, the field is nondeterministically initialized to any of the following:

- `null`
- a new object with uninitialized fields
- a reference pointing to any of the previously created objects of the same type (or subtype)

Primitive fields are always initialized to a new symbolic variable.

Method `_new_Node` in Program 4 (starting from line 8) is an example from such nondeterministic initialization. The method is called from the corresponding getter (*i.e.* `_get_next` starting from line 15). In `_new_Node`, the vector `v` contains the null object and all the objects created so far. The nondeterministic branching to select any item from `v`, or a completely new object, is on line 9.

Test data generation is launched by calling the method to be tested with an empty `this` object as argument. The empty object means an object with uninitialized fields. In the following, we will assume that `this` is the only reference argument, but other reference arguments would be handled similarly.

```
 1 public class Node {
 2   Expression elem;
 3   Node next;
 4   boolean _next_is_initialized = false;
 5   boolean _elem_is_initialized = false;
 6   static Vector v = new Vector();
 7   static {v.add(null);}
 8   Node _new_Node() {
 9     int i = Verify.random(v.size());
10     if(i<v.size()) return (Node)v.elementAt(i);
11     Node n = new Node();
12     v.add(n);
13     return n;
14   }
15   Node _get_next() {
16     if(!_next_is_initialized) {
17       _next_is_initialized=true;
18       next = Node._new_Node();
19       Verify.ignoreIf(!precondition());//e.g. acyclic
20     }
21     return next;
22   }
23   Expression _get_elem() {
24     if(!_elem_is_initialized) {
25       _elem_is_initialized=true;
26       elem = new SymbolicInteger();
27       Verify.ignoreIf(!precondition());//e.g. acyclic
28     }
29     return next;
30   }
31   Node swap() {
32     if (_get_next() != null &&
33        _get_elem()._gt(_get_next().get_elem())) {
34       Node temp = _get_next();
35       _set_next(temp._get_next());
36       temp._set_next(this);
37       return temp;
38     } return this;
39   }
40 }
```

Program 4: Excerpts from an annotated program

```
public class Node {
  int elem;
  Node next;

  Node swap() {
    if ( next != null && elem > next.elem ) {
      Node temp = next;
      next = temp.next;
      temp.next = this;
      return temp;
    }
    return this;
  }
}
```

Program 5: Example program

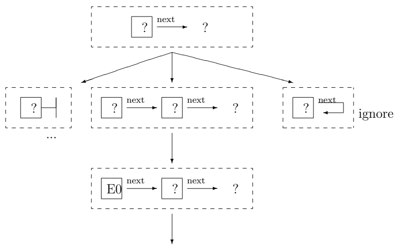

Figure 2: Excerpts from the symbolic execution tree of the Program 4, quoted from [12].

A problem is that lazy initialization can lead into illegal input structures. Thus, therefore a conservative class invariant is required. The invariant is implemented as a method that can determine if a (partially) complete object graph can be completed into a legal one. Actually such a precondition for each method separately would be sufficient. However, if an invariant can be defined, it can be used with all the methods of the class. Execution will backtrack if the invariant does not hold after the lazy initialization (see line 19 in Program 4).

Program 4 is the annotated version of Program 5. The example is quoted from Khursid *et al.* [12] with the annotation format quoted from Visser *et al.* [21]. The `precondition` method, which is not shown, is the class invariant that would return false if there is a loop in the list.

A partial symbolic execution tree of the program is provided in Figure 2. Only some first branches from the tree are taken into the figure. A question mark "?" inside a box is for an uninitialized value (*i.e.* elem field), but otherwise stands for an uninitialized reference (*i.e.* next field). In the initial state, the object for which the swap is called is created, but the fields (*i.e.* elem and next) are uninitialized. The figure demonstrates how new objects (*i.e.* list nodes and data objects in nodes) are created by the lazy initialization as the execution goes on. The first lazy initialization results from line 32 (Program 4). Evaluating "`_get_next() != null`" will result in the lazy initialization of the next field. The next will be initialized to any of the three possible cases, as illustrated by the first two rows of the figure.

What lazy initialization with symbolic values actually does, is generating the symbolic execution tree of the program. If the tree is finite, the approach will find all the leaf nodes of the tree, and therefore generate a test set with maximal path coverage [8]. However, if $\textsc{sym}(\mathcal{P})$ is infinite, the test data generation process does not terminate. One possible solution is to modify the JPF virtual machine so that only paths up to given length are checked. Another possibility is to set an upper limit for structure sizes in the class invariant. However, deriving actual test data from partially initialized object graphs is still an open problem. The constraint solver behind JPF will instantiate all the symbolic variables, but the unknown references are the problem. A simple solution is to make unknown references pointing to a special node called "unknown". Thus, graphs are not actually completed, but this should not be a problem because references pointing to "unknown" are not to be used as long as the program to be tested and the program to be used in the test generation are the same.

3. OUR APPROACH

Whereas the previous section was about related research of others, this section is about our own contributions to visualize test data and refine the symbolic execution based test data generation approaches of Section 2.

3.1 Visualizations for Abstract Feedback

Conceptually, in the approach we are now proposing, the outcome of automatic test data generation is not only a test set, but a set of *test patterns*. Each test pattern defines test data that are somehow similar. Test pattern is a kind of opposite to test set because the latter contains different test data in order to provide good test coverage. A possible grouping criteria for test patterns is that the execution paths in the program are identical. In detail, a test pattern consists of a single *test schema* and possibly several test data derived from the schema. All the test data in the same test pattern are derived from the schema of the pattern. Finally, the test set is obtained by selecting arbitrary test data from each test pattern.

The schema is an object graph with two special features: 1) object references can be unknown and 2) symbolic expressions are used for primitive fields. In addition, the schema has constraints related to the symbolic expressions.

The schemas will be used to demonstrate tests on a higher abstraction level when compared to the actual test data. To understand the use of schema in the feedback, let us consider test schema $\mathbf{s}$ and test data $\mathbf{t}$ derived from $\mathbf{s}$. Instead of exact feedback saying $\mathcal{P}(\mathbf{t})$ fails (or works correctly), we will provide abstract feedback like "$\mathcal{P}(\mathbf{s})$ fails (or works correctly)". However, the oracle of the automatic assessment is based on investigating $\mathcal{P}_{\text{specification}}(\mathbf{t}) = \mathcal{P}_{\text{candidate}}(\mathbf{t})$, as in the traditional approach. Figure 3 illustrates this process and the related terminology.

Because test schema is an object graph with some constraints, it can be easily visualized. Figure 4, for example, provides visualizations from partially initialized object graphs of a delete method in a binary search tree. These example schemas were obtained after generalized symbolic execution with lazy initialization. Figure 5, on the other hand, gives examples from the possible test data that can be derived from the schema of Figure 4.

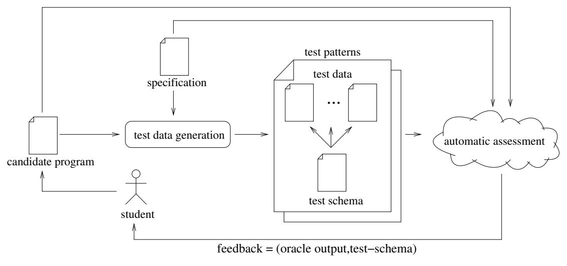

Figure 3: The process of creating feedback for students and some related terminology.

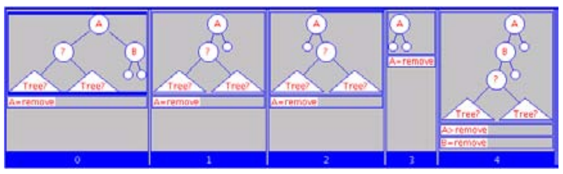

Figure 4: Excerpts of different input structures for the delete method of binary search trees

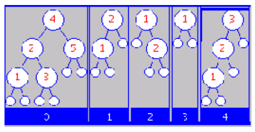

Figure 5: Examples of instantiated input structures from schemas in Figure 4

There are four types of nodes in the schema visualization: null nodes (small empty circles), nodes that are known to exist, but the data of the node is newer used (circles with ?), nodes with a data element that is used by the algorithm (circle with a letter), and nodes that represent a reference that is not used by the method (triangular nodes). In nodes where the data is used, keys inside nodes are symbolic variables and constraints over those variables are also provided.

3.2 Without Annotation

When deriving tests from students' programs, the manual annotation is not acceptable. This is because in automatic assessment the test data is generated on-the-fly, whenever a student submits a solution. Two techniques to remove the need of annotation in different use cases will be introduced: 1) use of the Comparable interface 2) A common upper class to a candidate program and the specification, called a probe.

3.2.1 Comparable Interface

Use of the Comparable interface can remove the need of replacing int type with SymbolicInteger. For example, let us assume a container implementation without primitive fields and where the data stored implements the Comparable interface. The interface is a standard Java interface used with objects having a total order. If the argument type in insert and remove methods of the container is Comparable, we can introduce the symbolic execution by using a special object that is comparable and hides the symbolic execution (Program 6). Moreover, students do not need this special class because they can test their container implementations, for example, with Integer wrappers.

```
public class ComparableSymbolicInt extends
     SymbolicInteger implements Comparable {
  public int compareTo(Object other) {
    ComparableSymbolicInt o = (ComparableSymbolicInt)
         other;
    if (this._LT(o)) return -1;
    else if (this._GT(o)) return 1;
    else return 0;
  }
}
```

Program 6: Definition of a comparable type that can hide symbolic execution so that programmers should not need to care about that.

The drawback of the Comparable approach is that in the comparison the execution will split into three when comparing SymbolicIntegers would split the execution into two. We will come back to this in Section 4.3.

3.2.2 Probes to Hide Invariants

Probes contain the specialized getters and setters of the lazy initialization as well as the class invariant. This makes it possible that those are not needed in classes derived from the probe. Thus, on-the-fly test generation from students programs is basically possible. The behavior of the getters and setters can be controlled so that lazy initialization can turned on and off.

The obvious limitation of probes is that new fields cannot be declared in a class derived from the probe. If new fields would be declared, lazy initialization of those would not be possible. This is mainly because the invariant method (declared in the probe) cannot say anything about the new fields. Probes can be easily applied to exercises dealing with exactly defined data structures (*e.g.* implement the red-black-tree or implement the AVL-tree). The problem is how to handle more open assignments where the structure of the class can vary from solution to solution. Most likely, the probe approach cannot be used in such cases.

4. DISCUSSION

Three fundamentally different approaches of using JPF were described in Section 2.2: 1) explicit method sequence exploration 2) symbolic method sequence exploration, and 3) generalized symbolic execution with lazy initialization. In Section 3.2, we introduced two new approaches: the Comparable interface and probes. The new techniques are designed to help test data generation directly from students' candidate programs. New techniques can be combined with previous techniques and Figure 6 summarizes the resulting six[2] different test data generation approaches.

On the upper level, we have separated techniques between *method sequence exploration* and *lazy initialization*. On the second level, symbolic execution is used with lazy initialization but it can also be used in (symbolic) method sequence exploration. Symbolic execution and lazy initialization both need different types of annotations. New techniques we have developed are hiding these annotations from users. The Comparable interface answers to the challenge of symbolic execution and probes to the lazy initialization specific problems.

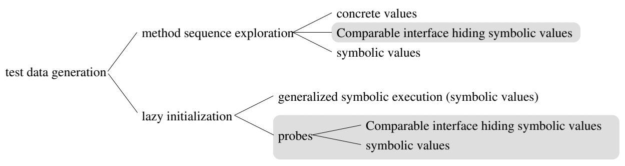
test data generation
method sequence exploration
concrete values
Comparable interface hiding symbolic values
symbolic values
lazy initialization
generalized symbolic execution (symbolic values)
probes
Comparable interface hiding symbolic values
symbolic values

Figure 6: Different test data generation approaches under discussion: new techniques introduced in this work are on gray background, whereas techniques on white background are from the related (previous) research.

4.1 Preparative Work

Table 1: Evaluating the annotations needed

technique		annotation
Method sequences	with concrete	none
	with comparables	none
	with symbolic	automatic
Lazy initialization	generalized symbolic	semiautomatic
	probes with comparables	none
	probes with symbolic	automatic

The preparative work is evaluated based on the amount of annotations required. The scale includes values *none*, *automatic*, and *semi-automatic*. None means that absolutely no annotation is needed, automatic means that the annotation process can be automated, and semiautomatic means that the annotation process can be partially automated, but substantial amount of manual work is still needed. Table 1 summarizes our observations in this category.

[2]Not all combinations are reasonable.

Method sequence exploration requires some annotation if symbolic arguments are not hidden behind the Comparable interface. However, the annotation process can easily be automated as variables of int type are only replaced with SymbolicInteger variables and operations between integers are replaces with method calls. Visser *et al.* [21] have already described a semiautomatic tool for the task. Actually the tool can also construct additional fields needed in generalized symbolic execution with lazy initialization as well as getters and setters for fields. Use of fields are also replaced with getter calls and definitions (*i.e.* assignments) with calls to corresponding setters. The only task in the tool that is not automated is the type analysis.

The annotation that cannot be automated in generalized symbolic execution with lazy initialization is the construction of invariants or preconditions. However, probes can be used to hide invariants and other needs of annotation – just like Comparable hides simple use symbolic integers. The framework can provide support for common data structures and algorithms. For more exotic classes, a teacher can implement the probe for students. In both cases, if a probe is available, handmade annotations are not needed.

4.2 Generality

Table 2: Evaluating the generality

technique		generality
Method sequences	with concrete	*****
	with comparables	**
	with symbolic	****
Lazy initialization	generalized symbolic	****
	probes with comparables	*
	probes with symbolic	***

Generality is about what kind of programs can be used as a basis in test data generation. Table 2 gives relative ranking between techniques – more stars in the figure indicate that there are more situations in which the technique can be applied.

Method sequence exploration has practically no limitations and is therefore ranked at the highest place. The other techniques are first ranked according to the comparable *vs.* symbolic classification. Use of symbolic objects is considered a more general approach when compared to Comparables. The secondary classification criteria is the use of probes. If probes are not needed, it is considered more general when compared to cases where the program is built on probes.

In the concrete method sequence exploration, all the possible operations with arguments (*i.e.* integers) are directly supported. Bit level operations are also supported and data flow can go from input variables to other methods (*e.g.* to libraty methods that are difficult to annotate). The symbolic execution framework of JPF does not support bit level operations. In addition, data flow from test data to other methods is problematic in symbolic execution. Such an attempt would require the same preparative work for other methods, as well. For library methods, this might be extremely tricky. However, limiting the program to Comparables is considered a more significant drawback when compared to the limitations of symbolic integers. There are many practical examples when a simple program needs integer arguments, and the computation cannot be performed with comparable arguments only.

Probes can also limit the generality. A new probe is needed for every possible data structure, which limits the number of supported programs.

4.3 Abstract Feedback

Table 3: Evaluating the abstractness of schemas

technique		abstractness
Method sequences	with concrete	*
	with comparables	**
	with symbolic	**
Lazy initialization	generalized symbolic	***
	probes with comparables	***
	probes with symbolic	***

According to Mitrovic and Ohlsson [15], too exact feedback can make learners passive and therefore abstract feedback should be preferred. Correspondingly, on introductory programming courses at the Helsinki University of Technology, we have observed that exact feedback (*i.e.* "program fails where $a = 2, b = 4$") guides some students to fix the counter example only. After "fixing the problem", the candidate program might work with $a = 2$ and $b = 4$, but not with other values $a < b$.

In this category, the evaluation is based on how much test data can be derived from the same test schema. In other words, how general is the schema. All the described approaches have a property that executions leading into two different execution paths cannot be derived from the same schema. Table 3 gives the relative ranking between techniques – more stars in the figure indicate that the schemas are more general.

Concrete method sequence exploration is the least abstract method because the schema and test data are the same. Lazy initialization is the most abstract approach as test schemas with it are only partially initialized object graphs. For each partially initialized symbolic graph, there are (several) symbolic graphs that can be obtained through method sequence exploration.

Another aspect related to the abstractness of schemas is redundancy. We have defined schemas so that all the test data derived from a single schema will lead into identical execution paths. However, it is possible that there are several schemas stressing one path only. This is what we call redundancy. Therefore, the more abstract the schema is, the less redundant it is.

A reason why the concept of redundancy is interesting is that even with the most abstract approaches some redundancy exists. The extra branching, and therefore redundancy, that the Comparable interface brings was described in the previous chapter. When comparison of symbolic integers has two possible values (a < b is either true or not) the comparison of Comparable objects had three possible outcomes (less than, equal, and greater than).

Nondeterministic branching in lazy initialization will also add extra branching to the program. Let us think about binary search tree delete operation. If the node to be deleted has two children, the minimum from the right subtree will be spliced out as in Program 7 which is an excerpt from the delete routine. Both input structures in Figure 7 are obtained through the lazy initialization with probes. The node to be deleted is A in the both cases. In both cases, B is the smallest value in the right subtree of A. Thus, B is spliced out, by setting the link (originally pointing to B) to the right child of B. The right child of B is accessed. As a consequence, it is initialized to `null` or a new object. Because the `right` pointer in A is simply set to the right child of B, the execution is the same regardless of the value.

```java
if( node.getLeft() != null && node.getRight() != null
     ) {
  BSTNode minParent = node;
  BSTNode min = (BSTNode)node.getRight();
  while (min.getLeft() != null) {
    minParent = min;
    min = (BSTNode)min.getLeft();
  }
  node.setData(min.getData());
  if (node == minParent)
    minParent.setRight(min.getRight());
  else
    minParent.setLeft(min.getRight());
}
```

Program 7: Excerpts from the binary search tree delete routine

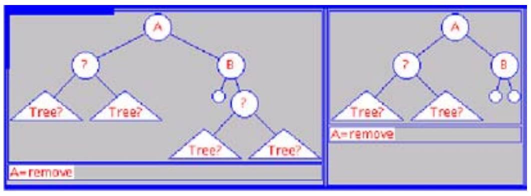

Figure 7: Two input data for the binary search tree leading to identical execution paths.

The same problem of extra branching in lazy initialization is present whenever branching does not depend on the initialized values. On the other hand, creating tests for such boundary cases (*i.e.* `null`s) might reveal some bugs that would otherwise be missed.

5. CONCLUSIONS

The work presents a novel idea of extracting test schemas and test data. Test schema is defined to be an abstract definition from where (several) test data can be derived. The reason for separating these two concepts is to provide automatic visualizations from automatically produced test data and therefore from what is tested.

On a concrete level, the work has concentrated on using the JPF software model checker in test data generation. Known approaches of using JPF in test data generation (*i.e.* concrete method sequence exploration, symbolic method sequence exploration, and generalized symbolic execution with lazy initialization) have been described. In addition, new approaches have also been developed:

Use of Comparable interface that removes the need of annotation in the previous symbolic test data generation approaches. The drawback of the approach is that only programs using comparables can be used.

Use of probes to remove the manual invariant construction needed by the lazy initialization.

Both new approaches are also a step from model based testing towards test creation based on real Java programs. Automatic assessment of programming exercises is not the only domain where the results of this work can be applied. Other possibilities are for example:

- Tracing exercises is another educational domain where the presented techniques can be directly applied. In tracing exercises test data and algorithm are given for a student. The objective is to simulate (or trace) the execution (*e.g.* [14]). The problem of test adequacy (*i.e.* providing test data for students) is the same as addressed in this research.
- Traditional test data generation can also benefit from our results. We believe that the idea of hiding the symbolic execution behind the Comparable interface is interesting. Extra branching resulting from the Comparable construction is not that bad, because it is nearly the same as boundary value testing (*e.g.* [9]). Instead of creating one test data for a path with the constraint $a \leq b$, two tests are created: $a = b$ (*i.e.* the boundary value test) and $a < b$.

As a summary, interesting concepts and techniques to make automatic test data generation more attractive in teaching and especially automatic assessment are presented. Results can be reasonably well generalized and applied on other contexts than automatic assessment of programming exercises. However, the work is the first step to bring formally justified test data generation and education closer to each other.

Acknowledgements: This article is based on my masters thesis work and therefore I thank my thesis instructor Ari Korhonen and supervisor prof. Lauri Malmi.

6. REFERENCES

[1] K. Ala-Mutka, T. Uimonen, and H.-M. Järvinen. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3:245–262, 2004.

[2] C. Artho, D. Drusinksy, A. Goldberg, K. Havelund, M. Lowry, C. Păsăreanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In *Proceedings of Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop*, volume 2589 of *LNCS*, pages 87–108. Springer-Verlag, 2003.

[3] M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, and M. Veanes. Validating use-cases with the AsmL test tool. In *Proceedings of 3rd International Conference on Quality Software*, pages 238–246. IEEE Computer Society, 2003.

[4] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. M. Zin. Ceilidh: A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*, Vienna, Austria, 1993.

[5] G. Brat, W. Visser, K. Havelund, and S. Park. Java PathFinder - second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification, Chicago, Illinois*, July 2000.

[6] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[7] D. Coward. Symbolic execution and testing. *Inf. Softw. Technol.*, 33(1):53–64, 1991.

[8] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

[9] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[10] D. Jackson and M. Usher. Grading student programs using assyst. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339, New York, NY, USA, 1997. ACM Press.

[11] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[12] S. Khursid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings 9th International Conference on Tools and Algorithms for Construction and Analysis*, volume 2619 of *LNCS*, pages 553–568. Springer-Verlag, April 2003.

[13] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[14] A. Korhonen, L. Malmi, and P. Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of the 3rd Annual Finnish/Baltic Sea Conference on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

[15] A. Mitrovic and S. Ohlsson. Evaluation of a constraint-based tutor for a database language. *International Journal of Artificial Intelligence in Education*, 10:238–256, 1999.

[16] C. S. Păsăreanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of 11th International SPIN Workshop*, volume 2989 of *LNCS*, pages 164–181. Springer-Verlag, 2004.

[17] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 133–136, Canterbury, UK, 2001. ACM Press, New York.

[18] L. Salmela and J. Tarhio. ACE: Automated compiler exercises. In *Proceedings of the 4th Finnish/Baltic Sea Conference on Computer Science Education*, pages 131–135, Joensuu, Finland, October 2004.

[19] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *Proceedings of the sixth conference on Australian computing education*, pages 317–325. Australian Computer Society, Inc., 2004.

[20] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng*, 10(2):203 – 232, April 2003.

[21] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM Press, 2004.

[22] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.