INSTITUT FÜR SOFTWARE & SYSTEMS ENGINEERING
Universitätsstraße 6a D-86135 Augsburg

Specification-based and Concolic Testing for Games

Alexander van Renen

Masterarbeit im Elitestudiengang Software Engineering

SOFTWARE ENGINEERING
Elite Graduate Program

INSTITUT FÜR SOFTWARE & SYSTEMS ENGINEERING
Universitätsstraße 6a D-86135 Augsburg

Specification-based and Concolic Testing for Games

Matrikelnummer:	1280500
Beginn der Arbeit:	4. März 2015
Abgabe der Arbeit:	31. August 2015
Erstgutachter:	Prof. Dr. Alexander Knapp
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Betreuer:	Prof. Dr. Alexander Knapp

SOFTWARE ENGINEERING
Elite Graduate Program

I assure the single handed composition of this master's thesis only supported by declared resources.

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 13. Dezember 2015 Alexander van Renen

Abstract

We compare and evaluate three tools for automated test input generation for Java programs. With manual software testing being a labor intense process, these tools can significantly reduce the amount of work for writing tests. We present two specification-based tools, Korat and TestEra, and one concolic tool, JCute. After explaining each tool individually, an analysis of their performance and usability for different testing scenarios is given. A key contribution of this work lies in the improvements made to Korat. By fixing several bugs and rewriting parts of its algorithm, we made it possible to use Korat for a wider range of projects. Due to lacking features and usability issues in the other evaluated tools, Korat appears to be the most favorable tool at the moment: Our experiments show that Korat is able to generate all possible instances of a data-structure up to a moderate size. For binary trees it is possible to generate all instances with up to 13 nodes in under a minute: This corresponds to over a million trees.

Kurzfassung

Wir vergleichen und bewerten drei Tools zur automatischen Testgenerierung für Java-Programme. Diese Tools können den arbeitsintensiven Prozess des manuellen Schreibens von Softwaretests deutlich verbessern. Wir stellen zwei spezifikationbasierte Tools, Korat und TestEra, und ein concolisches Tool, JCute, vor. Jedes Tool wird zunächst individuell vorgestellt. Im Anschluss wird ihre Performance und ihre Usability untereinander in verschiedenen Test-Szenarios verglichen. Einer der wichtigsten Beiträge dieser Arbeit besteht in den Verbesserungen, die an Korat vorgenommen wurden. Durch das Beseitigen einiger Fehler und durch Verbesserungen am Algorithmus ist es nun möglich, Korat für ein umfangreicheres Anwendungsgebiet zu nutzen. Aufgrund des Fehlens zentraler Features in den anderen untersuchten Tools scheint Korat derzeit das am besten entwickelte Tool zu sein: In unseren Experimenten ist zu sehen, dass Korat in der Lage ist, alle möglichen Instanzen einer Datenstruktur bis zu einer moderaten Größe zu erzeugen. Bei binären Bäume ist es möglich, alle Instanzen mit bis zu 13 Knoten in unter einer Minute zu generieren: Das sind über eine Million Bäume.

Contents

Abstract **4**

1 Automated Testing in Software Development **7**
1.1 Manual Software Testing 7
1.2 Specification Based Testing 10
1.3 Concolic Testing 14

2 Korat **18**
2.1 Description 18
2.2 Algorithm 23
2.3 Modifications 37

3 TestEra **51**
3.1 Description 51
3.2 Transformations 55
3.3 Limitations 59

4 JCute **63**
4.1 Description 63
4.2 Algorithm 69

5 Experiments and Evaluation **77**
5.1 Environment 77
5.2 Sample Project: Chelone 78
5.3 Experiment: Expression Tree Serialization 78
5.4 Experiment: Collision Detection 83
5.5 Experiment: Item Bonus Calculation 87
5.6 Experiment: Input Queue Synchronization 88

6 Conclusions and Future Work **91**
6.1 Korat 91
6.2 TestEra 93
6.3 JCute 94
6.4 Overall 95

1 Automated Testing in Software Development

Testing is a key component of software development. With formal verification techniques mostly being used in high risk environments, testing remains the primary way for assuring the quality of a product in most companies. Next to functional correctness, testing can also serve as a documentation, give confidence in software components and help with keeping the code base modular [36, 27].

Nevertheless, developers often see testing as a burden, as it slows down their progress. This is undoubtedly true for the immediate presence: It does take time and therefore money to write tests. Even so this effort is necessary for the product to succeed, the amount of required manual work can be significantly reduced with automated testing [10].

In addition, writing tests manually is not only a labor intense process, in general it is very difficult (if not impossible) to test all possible inputs. Even testing all inputs up to a given size is difficult, as the developer has to write down all inputs manually or invent an algorithm to iterate over possible inputs. One way to solve this problem for some kinds of tests is to use specification-based or concolic testing.

1.1 Manual Software Testing

As an example, consider an algorithm for traversing a binary tree. These kinds of algorithms are often implemented using a visitor pattern [18]. They are used to walk through all nodes of a tree in a specific order. They can be used when working with syntax trees to analyze or interpret the language. Another use case is a simple Java program, which prints out or inspects the content of a tree by iterating over each node.

Next to reading a binary tree, another example is to modify the tree. Depending on the tree, these algorithms can be quite difficult. An interesting candidate here is the red black tree [20][1]. Its good runtime characteristics makes it a prime candidate for the implementation of sorted sets and maps in libraries. For example in the JDK, a red black tree is used to implement the `TreeMap` [4] and `TreeSet` [5] data-structures. They are also often used in the kernel of operating systems and in other standard libraries like C++ or C#. This wide usage makes it extremely important that the operations (like insert, remove or iterate) are well tested.

[1] Originally known as symmetric binary B-Tree [9].

```
1  class TreeSetTests {
2     @Test
3     public void testTreeSetInsert1() {
4        // Generate a tree
5        TreeSet<Integer> treeSet = new TreeSet<>();
6        treeSet.add(0);
7        treeSet.add(1);
8        treeSet.add(2);
9        treeSet.add(3);
10
11       // Pre condition
12       assertTrue(treeSet.contains(2));
13
14       // Test
15       treeSet.remove(2);
16
17       // Post condition
18       assertFalse(treeSet.contains(2));
19    }
20
21    // ... more tests
22 }
```

Listing 1.1: A test for remove function in TreeSet.

Section 1.1 shows a typical test for a remove function on a tree structure: A developer first has to write code, which creates a tree object with the respective properties (line 5-9). After that the code has to invoke the algorithm to be tested on the tree (line 15) and then verify the result (line 18).

Even though this code seems straightforward, there are a number of problems with this attempt:

- **Modifiability**
  It is not clear how the tree looks like and which kind of node is being removed in the test case. A good test suite for a remove function should at least exercise more structurally different node types, for example: A root node, a leaf node, a node with one child and a node with two children should be removed from the tree in order to increase the coverage of the remove method. This becomes difficult, as the tester does not know how the tree looks like after simply inserting nodes.

  To solve this problem, the tester needs to know the structure of the created tree. With that it is possible to remove a node of a specific structural type. This could be done in two ways: First, one could create the tree structure directly by setting the internal fields, or second, the insert method calls could be ordered in such a way that the resulting tree has the wanted structure.

  Either approach gives the tester knowledge of the underlying tree structure and allows him to remove a specific node. Thus all structurally different remove operations

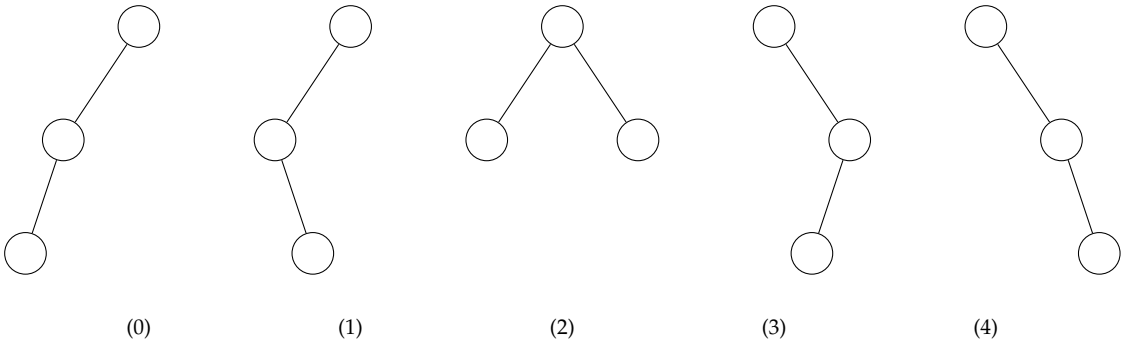


Figure 1.1: All possible binary trees with exactly 3 nodes.

can be tested. But both solutions have the same problem: They make the tests dependent on the internal implementation of the tree. Any changes there would break the tests or lead to a test suite which still runs cleanly, but does not validate the removal of certain node types.

- **Scalability**
  Depending on the tree, there are usually more interesting cases for the remove function than the earlier mentioned structurally different four node types. In the red black tree example, it could be helpful to generate trees which need different rotations to re-balance the tree during the remove operation. The function:

  $$f(n) = \sum_{i=0}^{n} \frac{(2n)!}{(n+1)!n!}$$

  describes the number of binary trees with up to n nodes. The formula is derived in section 2.2.1.1. It shows that it is practically impossible to manually write down all test cases for the remove function for all binary trees with a bigger than tiny node size. With only up to 5 nodes there are already 65 possible trees. Writing down tests for 65 different trees generates huge amounts of test code and is very time consuming.

- **Completeness**
  Besides the huge number of possible binary trees for a larger node size, it is difficult to verify that all 65 test cases generate different trees. Figure 1.1 shows all trees with exactly 3 nodes. Developers are often under a lot of pressure to create new features. With time being limited, developers often have to skip some of the cases. This leads to an incomplete test suite even for the small tree size.

A solution to these problems seems rather straightforward: An algorithm which generates binary trees. The problem is that these kinds of algorithms are not trivial to write.

Therefore, one needs to test the generation algorithm, in order to check that all possible trees are generated. In addition, the tester needs to implement such an algorithm for every data-structure which ought to be tested.

Therefore, one should not try to reinvent the wheel for every data-structure by creating custom generators, but rather use a tool which is able to generate any data-structure up to a given size using a specification. This would allow the developer to test the algorithm for all possible inputs, without the manual labor of creating them or an algorithm for generating them every time.

This is exactly what tools for specification based testing help the developer with: Given a formal specification for the input, the tool is able to generate all instances which are conform with the specification.

1.2 Specification Based Testing

Next to manual software testing, there are several automated ways to test software. Using automated tools can significantly improve the productivity when testing software. This section introduces one of these techniques: Specification Based Testing.

Specification based testing is a form of functional testing [10, 11], which is also known as black box testing. The idea is to look at the program or function which is to be tested as a black box: The tester or the testing tool is only interested in the input and output. How the function works internally is irrelevant as long as the observable output is correct.

This stands in contrast to structural testing (or white box testing), where the tester or the testing tool uses the implementation of the function under test to derive test cases.

1.2.1 On Specifications

The goal of specification based testing is to compare the specification of a function[2] with the behavior of its concrete implementation. Any differences imply a failure, either in the specification or the implementation or both. The absence of failures shows the compliance of the implementation with the specification for the tested subset of the input domain. However, it does not, as always in testing, prove that the function is correct. Only examining the complete input domain would prove that the function is fully conform with the specification.

As already hinted, a specification is not necessarily perfect, either. It is simply a model of the desired system in a different format or language written by another (or the same) developer. Therefore, it may also be incorrect or incomplete. In addition, there is a chance to over- or under-specify the system. And even when assuming a correct and complete specification, still it may not be what the customer or system actually needs. All specification based testing can do is to show the compliance of the implementation with the

[2] From now on we will only refer to single functions. The same concepts can be applied to testing programs, components, modules and so on.

```
1  abstract class TreeSet<T> {
2
3     /**
4     * Removes the specified element from this set if it is present.
5     * More formally, removes an element {@code e} such that
6     * <tt>(o==null ? e==null : o.equals(e))</tt>,
7     * if this set contains such an element.  Returns {@code true} if
8     * this set contained the element (or equivalently, if this set
9     * changed as a result of the call). (This set will not contain
10    * the element once the call returns.)
11    *
12    * @param o object to be removed from this set, if present
13    * @return {@code true} if this set contained the specified element
14    */
15    public boolean remove(Object o) {
16       // ...
17    }
18 }
```

Listing 1.2: Informal method specification in Oracle's JDK 1.7.

specification.

There are two different forms of specifications. Automated tools require the second one:

- **Informal Specification**

  This form of specification includes anything reaching from contracts or emails to user requirements or meeting notes. For single functions the specification can also consist in the documentation or it can be implied by the function name in combination with the coding standard of the project.

  An informal specification is usually less precise and complete than a formal specification. It is written in natural language, thus no special skills are required to understand it, neither by the developer nor the customers.

  As an example consider the specification in section 1.2.1. It is taken directly[3] from the source code in Oracle's JDK 1.7. During development, most programmers would look here to learn what the method does.

  It is written using JavaDoc [32], which is widely used throughout the Java community. As shown in the example it is directly inserted into the source code, but it can be automatically extracted and transformed into a documentation[4]. JavaDoc is not a specification language, but only a formating language. The actual specification is still written in natural language.

  Besides not being readable by a computer, there are two smaller problems with this specific specification (section 1.2.1). First, parts of it are difficult to read (line 6) when

[3] Line breaks are modified and `@throws` clauses are removed.

[4] For example, the documentation for Oracle's JDK is generated using JavaDoc [5, 4].

```
1  abstract class TreeSet<T> {
2
3      /*@
4      @ ensures \result == \old(contains(o))
5      @       && !contains(o);
6      @*/
7      public boolean remove(Object o) {
8          // ...
9      }
10 }
```

Listing 1.3: Formal method specification for remove method using JML.

looking at it without a tool to parse the HTML codes. Second, it is redundant, because the return value is described twice (line 7-8 and line 13).

- **Formal Specification**

  These are written in a formal language like Z specifications [21, 42], JML [31, 30, 13] or OCL [43]. They are often expressed as pre- and post-conditions and invariants. Therefore, they can be processed, understood and worked with by a computer. This makes it possible to derive certain test cases from the specification automatically. Therefore, it's possible for a tool to check the compliance of the implementation and the specification. In turn, it is less intuitive to write and harder to understand for people without training in formal languages (in general especially customers).

  As an example, consider the specification in section 1.2.1. It is written in the Java Modeling Language (JML). In the first line of the specification (line 4) the `ensure` keyword tells JML that we are defining a postcondition. It can be read like this: "the method ensures that (1) its result is equal to the result of the call to `contains(o)` before its execution and (2) that after its execution the result of `contains(o)` is `false`".

  One advantage, compared to the informal specification, is its size: it only takes up 2 lines instead of the 9 lines needed for the informal specification in section 1.2.1. In addition, it is readable by an automated tool. For example, this specification could be directly used by Korat. The drawback is that it can not easily be understood by a developer unfamiliar with JML.

1.2.2 Generating Test Cases from Specification

This thesis focuses on formal specifications. Tools like Korat [12] or TestEra [35] can use a formal specification of a function to automatically generate valid inputs for the function. Usually they can not generate all possible inputs, as the input spaces of a function tend to be extremely large[5]. Hence, the tester has to define bounds on the input space. Figure 1.2

[5]Consider a function working on a 64 bit integer. There are already $2^{64} = 1.8e19$ possible inputs.

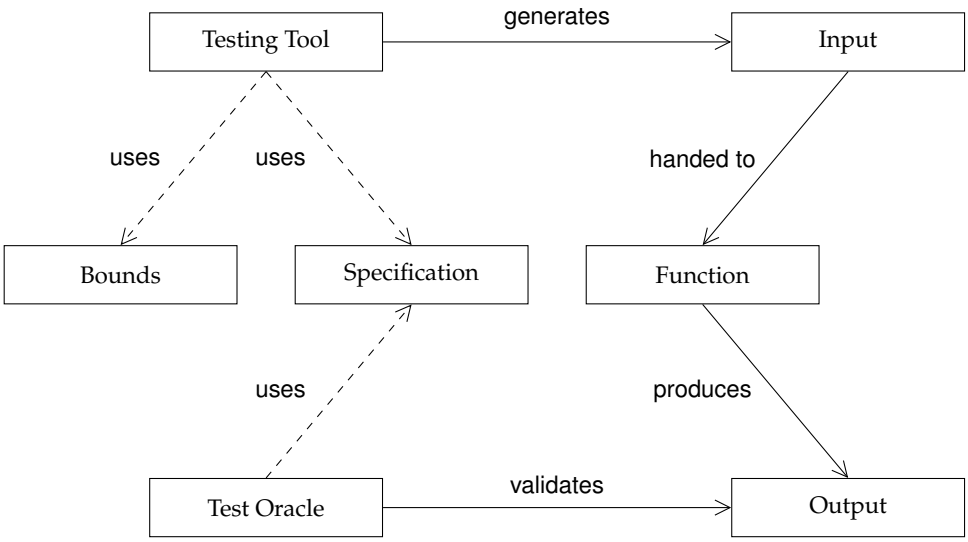


Figure 1.2: Automated specification based testing tools.

illustrates this process.

Using the *Specification* and the *Bounds*, the *Testing Tool* is able to generate a valid *Input* for the function under test. The generated *Input* is then passed to the *Function* under test. Its *Output* is then validated by the *Test Oracle* using the *Specification* again. The process is repeated for as long as the *Testing Tool* is able to find more valid inputs within the specified *Bounds*.

The usage of an automated tool to generate the input for the test function takes care of some of the problems with manual testing in section 1.1:

- **Modifiability**
  The tree is no longer constructed via method calls to the tree interface. It is usually (in case of Korat and TestEra) created by assigning the internal fields of the class directly. These tools guarantee that all possible instances are created. Therefore, the developer's tests run on all tree instances (within the bounds). Thus, all structurally different node types are covered.

  Just like when writing tests manually, this creates a dependency on the internal structure of the data-structure under test. In a specification, which is detailed enough to describe the structure to be generated, all internal fields and references of the object are specified. Hence, any change there will cause a problem.

- **Scalability**
  Using the automated tool, the developer does not have to write code for every tree instance anymore. After writing a single specification, one can generate as many

trees as one wants. The only limitation is the runtime, as generating thousands of trees takes some time.

In practice our experiments (section 5.3) have shown that it is possible to create all trees with up to 11 nodes (82500 trees) in about 3.3 seconds. In order to keep the test suite of a project fast, one could easily make this value configurable. This way, developers could run a fast test with only 5 nodes (65 trees) on their development machines. In addition, there would be a larger test suite with 10 or 11 nodes on their continuous integration server, which runs only every night.

- **Completeness**
  Tools like Korat and TestEra guarantee the exhaustive exploration of the entire search space. Hence, there is no risk that a developer overlooks certain cases. Using one of these tools, the development team can be sure that all cases have been tested.

Note: As the number of structures usually grows really fast, one is only able to generate structures up to a small size. Hence, it might be necessary to write additional tests, which exercise larger structures. However, our anecdotal experience has shown that most bugs are already present in relatively small structures and not only in trees with hundreds of nodes.

1.3 Concolic Testing

Another alternative to manual software testing is concolic testing. The word *concolic* is a combination of *conc*rete and symb*olic*. Similar to specification based testing, concolic testing aims to improve productivity by automating the labor intense process of manually generating inputs for test cases.

Concolic testing is a structural (white box) testing method: It does not view the method under test as a black box with only input and output, but looks at its inner workings. It analyses the code to gain additional knowledge and thereby improve the test case generation process.

For any method, the possible paths through the program can be visualized as a control flow graph. Figure 1.3 shows an example: The program shows an implementation of a comparison method between two integer values. The method returns -1, 0 or 1 if the first integer is less than, equal or greater than the second integer. On the right hand side of the code, the control flow graph is shown. Obviously, there are three different paths through this method.

The idea of concolic testing is to explore all possible paths, thus covering all possible outcomes for all conditions in the program. If there are loops or jumps involved, there can be a large number or even an infinite amount of possible paths. In addition, not all paths are always possible. Concolic testing aims to cover as many as possible.

To achieve this, a concolic testing tool combines concrete and symbolic testing principles:

```
1  int compare(int lhs, int rhs) {
2     if(lhs < rhs) {
3        return -1;
4     } else {
5        if(lhs == rhs)
6           return 0;
7        else
8           return 1;
9     }
10 }
```

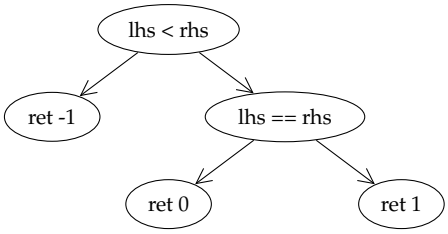


Figure 1.3: Control flow graph of a simple function.

- **Concrete**
  The method or program under test is simply executed. The code is not analyzed. The idea is to randomly generate the method's input parameters, run the program and hope to trigger a fault. The advantage of this approach is that it can be completely automated.

  One member of this group of tools is JCrasher [16], which does purely random test input generation. To make this process more efficient, a lot of tools try to guide the random search by observing the methods behavior. One example is DART [19], which can be used to test interfaces of data-structures. Starting with a single method call, it successively chains calls to the interface, trying to find a sequence to cause a fault.

  The main problem of these approaches lies in generating more complex data-structures. If the method under test expects a data-structure as an input, it first has to make sure that the generated input is valid, i.e. the invariants of the structure are fulfilled. The problem with randomly generating these structures is that it becomes very unlikely to find them. And therefore, even more unlikely to find different ones [38].

- **Symbolic**
  Most of the tools [17, 8] using this approach inspect the code itself. They abstract a formal model and use a model checker to check certain properties. In Java, these checks often test whether a null reference is dereferenced, an array is accessed out of bounds or a certain statement is reachable.

  There are two problems with this approach: First, as the tools are working on an abstract of the actual code, not every issue they find is necessarily a problem in the real program (no soundness). Second, for more complex algorithms, especially involving pointer arithmetic, it is often too hard to solve the constraints [40].

1.3.1 The Idea

In concolic testing, the worlds of concrete and symbolic execution are combined: The method under test is executed multiple times. For the first execution, the input values are chosen randomly. After that, they are derived from the previous execution. To achieve this, the concolic testing framework gathers constraints on the input variables at each branching point of the method. After the method is finished executing, these constraints are conjuncted. The resulting logical formula describes the characteristics of the input parameters for this particular path. All inputs satisfying the formula lead to the same path through the method. For the next iteration, parts of the formula are negated and a constraint solver is used to find concrete values which satisfy the new formula. The resulting set of input parameters lead to a different execution path.

In practice, this approach can achieve a high coverage within few iterations. The advantage, compared to specification-based testing, is that it mostly explores different paths through the method. Nevertheless, the usage of the constraint solver is still rather slow, as examined in chapter 5. Using concolic testing, one can avoid some of the problems with manual testing, which were introduced in section 1.1:

- **Modifiability**
  When asking how concolic testing handles changes on the tested data-structure, one first has to understand how concolic testing can be used to generate data-structures. Therefore, recall the previously introduced example in fig. 1.3: A concolic testing tool can systematically explore the three possible paths through the `compare` method. This feature can be used for generating data-structures: One simply generates values and inserts these into the data-structure via method calls[6]. The concolic testing framework will try to explore all different paths through the insert method by choosing different values. Thus, creating structurally different structures. Note that unlike the generation process in specification based testing, this approach does not guarantee the exhaustive exploration of the search space. It is highly dependent on the insert method.

  As an example, consider a binary tree structure for storing integer values. Trees with 4 nodes can easily be generated using the following code snippet:

```
1 int testBinaryTree(int a, int b, int c, int d) {
2    BinaryTree tree = new BinaryTree();
3    tree.insert(a);
4    tree.insert(b);
5    tree.insert(c);
6    tree.insert(d);
7
8    // Perform tests on the tree
9 }
```

[6] This assumes that the data-structure under test offers some kind of method to add values.

The concolic testing tool tries to explore all different paths through the `insert` methods, thus creating all structurally different trees. In this example, it is assumed that the `testBinaryTree` method is called by the concolic testing framework with generated values.

There is a big advantage of using the interface of the data-structure to generate it: It is independent of the internal structure and therefore any changes in the implementation will not cause any problems with the test suite. Only changes in the API of the data-structure could cause problems, but these kinds of changes are unlikely as they would also break the production code which is using the data-structure.

- **Scalability**
  As shown for the binary tree example, one can use concolic testing tools for automated test case input generation. In order for this to work, the data-structure that is to be tested only has to offer a method which can be used to construct it. This approach frees the developer from writing all test cases by hand.

  Due to the complex nature of constraint solving, it is not possible to generate as many structures as with specification based tools, like Korat. In practice, we were able to generate trees with up to 3 nodes (9 tress) in about 6.7 seconds, for more details see section 5.3.

- **Completeness**
  While the concolic testing tool is able to generate data-structures automatically, it does not guarantee an exhaustive exploration of all possible instances. Only these, which can be created by a sequence of constructing method calls, can be generated. Therefore, some instances could be overlooked.

2 Korat

Korat [12, 33] is a Java framework for specification-based testing, which was developed around 2004 at the MIT Laboratory for Computer Science. It can be used to automatically generate all non-isomorphic instances of a given data-structure or object graph up to a specified size.

In order to make this work, the user specifies a so called `finitization` of the data structure to be tested. The `finitization` specifies all objects which should be generated and how they are connected. Korat uses this information to generate all possible instances of the data structure. For each one, a Java predicate[1] (supplied by the user) is evaluated to determine the validity of the instance at hand. Not all possible instances of a data structure are valid, for example they could violate the data structure's invariants. Each instance which fulfills the predicate is passed to the user's test cases.

This system allows the user to easily iterate over all possible instances of a data structure without missing any. Therefore, all input variations can be tested without the labor intense task of manually writing down code to generate them. This exhaustive testing strategy naturally includes all corner cases of the input which could have been overlooked by a human tester.

In order to minimize the runtime, Korat monitors how the user code is using each generated instance of the data structure under test. Doing so, Korat is often able to discard large portions of the search space and thereby optimizes the runtime of the tests.

The specifications can be written in any language as long as the language can be automatically translated into Java predicates. In all experiments conducted in this thesis, Java is used to write the predicate method (usually called `repOK`). The ability to write the specification in the same language as the actual code has the advantage that it eliminates the need for programmers to learn new languages and in addition keeps source code files homogeneous regarding the programming language.

This chapter first gives an example guided explanation of how Korat works and how a developer can use it in section 2.1. After that the details of Korat's algorithm are explained in section 2.2. Lastly, section 2.3 provides a list of all changes made to Korat.

2.1 Description

As previously stated, Korat is able to automatically generate input data for test cases. These generated inputs can be used to basically execute any method. Each of these gen-

[1] A predicate is a method with no arguments (except the implicit `this` reference) returning a boolean

```
1 class BinaryTree {
2    static class Node {
3       Node left;
4       Node right;
5    }
6
7    Node root;
8 }
```

Listing 2.1: A simple binary tree.

erated inputs is just a normal Java object consisting of primitive and reference fields. The primitive values for the primitive fields can be generated and the reference fields can reference other generated objects.

As an example throughout this subsection we will use a binary tree as shown in Listing 2.1. The class `BinaryTree` has a field `root`, which is a reference to an object of type `Node`. In Addition, there is a static inner class which defines a `Node`. A Node has two child nodes: `left` and `right`. In this section we will show how Korat can be used to generate all non-isomorphic `BinaryTree` instances up to a specified size.

In order to make this work, Korat first calls a `finitization` method. This method defines the structure of the input: Which objects are involved and how are they connected. The implied set of elements is called a search space. Korat iterates through this space and invokes a `repOK` method on each element. This method checks if the given element is in a valid state. All valid elements are passed to the test method specified by the user.

In the binary tree example, one could try inserting or removing nodes from the tree. With Korat generating all possible trees, the insert or remove tests would be able to test for all node constellations: Nodes with no children, with one child, with two children, the root and so on.

2.1.1 Finitization

In order to generate test cases, Korat needs to know which objects it should generate and how they are connected. This information is defined by a `finitization` object. Korat will do an exhaustive walk through all elements of the implied search space. As these search spaces tend to be very large, Korat optimizes the search by pruning invalid and isomorphic elements (see section 2.2.3 and section 2.2.4).

For each field to be generated by Korat, the user has to assign a `FieldDomain` via the set-method of the `Finitization` object. A `FieldDomain` defines which values or references should be assigned to the particular field. For fields with a non-primitive type, it holds a set of `ClassDomains`, which all need to be assignable to the class of the `FieldDomain`. Each `ClassDomain` represents one class. This structure allows for fields with polymorphism by simply adding two different `ClassDomains` to one `FieldDomain`. Korat also allows for `FieldDomains` with primitive, array or enum types.

```
1  class BinaryTree {
2     // ...
3
4     static IFinitization finBinaryTree() {
5        IFinitization f;
6        f = FinitizationFactory.create(BinaryTree.class);
7
8        IClassDomain nodeCD = f.createClassDomain(Node.class, 3);
9        IFieldDomain nodeFD = f.createObjSet(nodeCD, true);
10
11       f.set(BinaryTree.class, "root", nodeFD);
12       f.set(Node.class, "left", nodeFD);
13       f.set(Node.class, "right", nodeFD);
14
15       return f;
16    }
17 }
```

Listing 2.2: Finitization method.

Listing 2.2 defines the `Finitization` for our `BinaryTree` example. We start off by creating a `Finitization` object (line 6). In order to do this, we need to tell the `FinitizationFactory` the class of our test input. An object of this class will always be passed to the test method and this class needs to define the `repOK` method. In our example we use the `BinaryTree` class[2]. After that we create a class domain for the three `Node` objects (line 8). This class domain is then wrapped in a field domain (line 9). The `true` argument for the `createObjSet` method tells Korat to include `null` values in the field domain.

Having defined all involved objects, we move on to defining their connections: The `Node` objects may be used for the `root` field of the `BinaryTree` class (line 11) and for the `left` and `right` field of the `Node` class (line 12 and 13).

2.1.2 The repOK Method

The second ingredient to supply Korat with is the `repOK` method, which defines constraints for the input. This method is either generated from the pre-conditions of the method under test or it is directly written in Java as a predicate.

In both cases, the method is called for each test input before Korat executes the actual test method. Its purpose is to check if all objects of the input are in a valid state and therefore represent a valid input, i.e. satisfy all pre-conditions and invariants. For a valid input, Korat will run the test method. Otherwise it will continue with the next element in the search space.

Listing 2.3 shows the `repOK` method for our `BinaryTree`. It will walk the tree via

[2] In other experiments conducted in this thesis, it often proved useful to use a wrapper class here.

```
1  class BinaryTree {
2     // ...
3
4     boolean repOK() {
5        if (root != null)
6           return root.repOK(new HashSet<Node>());
7        return true;
8     }
9     static class Node {
10       // ...
11
12       boolean repOK(HashSet<Node> alreadySeen) {
13          if (!alreadySeen.add(this))
14             return false;
15          if (left != null && !left.repOK(alreadySeen))
16             return false;
17          if (right != null && !right.repOK(alreadySeen))
18             return false;
19          return true;
20       }
21    }
22 }
```

Listing 2.3: repOk method.

depth first search and check that each node only occurs once. This makes sure that only acyclic trees are passed to the test method. It does so by creating a `Set` (line 6) and passing it to the root node. Each node will add itself to the set and check if it was already contained in the set (line 11). If the node was already in the set, it has been visited before and the tree is invalid.

2.1.3 Instrumentation

As described in section 2.1, Korat needs to monitor accesses to fields of the objects it has generated. Therefore, each time a field is accessed, the central test object, named `TestCradle`, is notified via a method call. The `TestCradle` stores which fields were accessed during the execution of `repOK` and uses this information for pruning the state space.

In order to get these notifications, Korat modifies the byte-code using the javassist[14] library. The instrumentation is done when a class is being loaded. The JVM loads a class only when it is being used (for example when it's being instantiated). This lazy class loading concept allows for a faster startup of the JVM and prevents wasting time on loading classes, which are never going to be used. Korat hijacks this system by replacing the default class loader with a custom one.

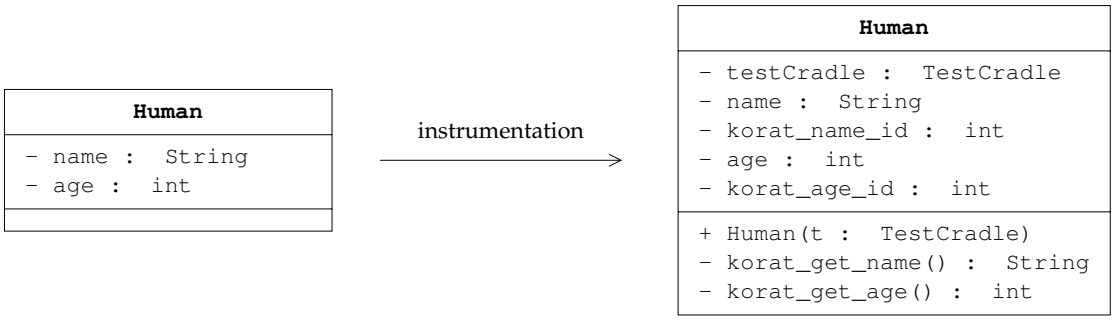


Figure 2.1: Instrumentation for non-array fields.

2.1.3.1 Monitoring Fields

For each non-array field of all classes Korat generates a special getter method and wraps all read access behind this new method. Inside the special getter method a notification method is called on the `TestCradle` object before the value of the field is returned.

To find out which field triggered a notification, Korat associates each field with a unique id. To achieve this, Korat further changes the byte-code and adds an id field for each field. The id field is initialized together with the value of the actual field when Korat creates a new object for the test input. Every time the field is accessed (using its special getter) this id is passed via the notification method to the `TestCradle` which is then able to determine which field is responsible for the notification.

The last thing missing in this system is a way to access the `TestCradle` object from within the generated getter methods. For this, Korat modifies the class once more and adds a reference field for `TestCradle` and a special constructor, which initialized the `TestCradle` field.

This system is illustrated by the two UML Class Diagrams in fig. 2.1. On the left hand side it shows the original class `Human` with a `name` and an `age` field. On the other side it shows the transformed class after the instrumentation. An id field and a getter was added for each field. In addition, there is a new constructor and a field for the `TestCradle`.

2.1.3.2 Monitoring Arrays

As an array represents not a single element but a collection of elements, it has to be handled differently. Korat needs to be able to monitor access to the array's length and to each element independently. Therefore, the developers of Korat chose to encapsulate each plain Java array in a so called `KoratArray`.

During the instrumentation phase (at class load time), Korat replaces all array fields by `KoratArray` fields. A new `KoratArray` class is generated for each type. In Figure 2.2, a `KoratArray` for the `Integer` type is shown. This class would be used to substitute a field of type `Integer[]`. In addition, Korat modifies the byte-code and replaces each array access by the corresponding methods in the `KoratArray`.

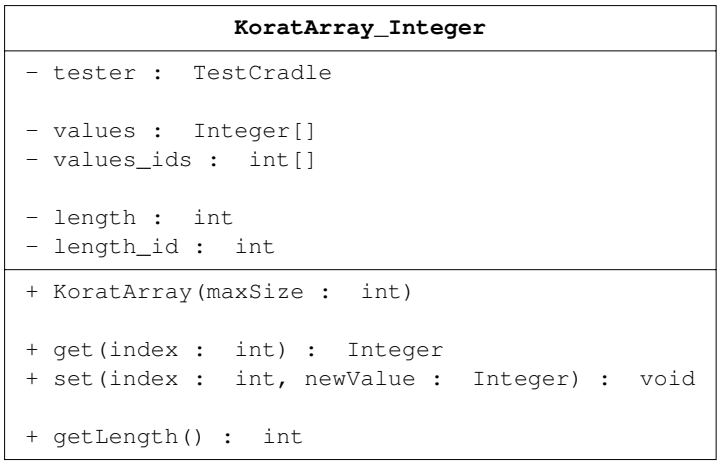


Figure 2.2: Instrumentation for non-array fields.

2.2 Algorithm

With these two methods, (`finBinaryTree` and `repOK`), Korat is able to generate all valid instances of a data-structure as an input for the user's test method. The problem is that search spaces tend to be rather large, as they grow exponentially with the number of involved fields. Usually, not all possible instances represent a valid state of the object. Whenever the ratio between valid instances and possible ones is low (i.e. the `repOK` method filters a lot of instances), Korat generates a lot of objects which are immediately discarded by the `repOK` method.

From a functional point of view, this is not a problem. But considering the large size of the search spaces, it is a performance problem, as it leads to a longer runtime for the test suite. To solve this problem, Korat uses pruning to discard large portions of the search space and an isomorphism detector to filter isomorphic objects, which do not differ structurally, but only have different object ids[3]. Together, these two optimizations limit the number of explored instances (the ones passed to `repOK`) significantly.

The following sections are structured as follows: Section 2.2.1 shows the optimization potential for the pruning and isomorphism filter by deriving formulas for the number of potential and explored instances for binary trees. After that, the naive algorithm for exploring the search space is explained in section 2.2.2 and then extended by the pruning (section 2.2.3) and isomorphism (section 2.2.4) filter.

2.2.1 Optimization Potential

To calculate the potential for these optimizations we need to compare the number of possible instances with the number of valid instances. We can describe these in the following

[3]Korat's notion of isomorphism is defined in section 2.2.4.

two functions:

- $f(n)$ The number of valid trees with up to n nodes. As the nodes are not distinguishable, this function does not include isomorphically equal trees. Therefore, it shows the number of trees which are actually generated by Korat after both filters.
- $p(n)$ The size of the search space, i.e. the number of valid and invalid trees with up to n nodes. This function includes all possible assignments for all fields.

2.2.1.1 Deriving $f(n)$

In our `BinaryTree` example, the only constraint expressed in the `repOK` method is that a given node may only occur once in a tree. The number of binary trees with exactly n nodes can be calculated with the formula $\mathcal{C}(n)$. This sequence is known as the *Catalan numbers* [1]:

$$\mathcal{C}(n) = \frac{(2n)!}{(n+1)!n!}$$

To get the number of binary trees with up to n nodes $f(n)$, it is simply a matter of forming the sum from 0 to n. This formula $f(n)$ represents the number of trees which are passed to the user's test function for a given node count.

$$f(n) = \sum_{i=0}^{n} \mathcal{C}(n)$$

2.2.1.2 Deriving $p(n)$

The size of the search space can be calculated by counting all possible instances. For this, one simply has to multiply the cardinality of the field domain (i.e. number of possible values for the field) for all fields. Let $|f|$ be the cardinality of field f and F the set with all fields references in the `finitization`. Then the following expression describes the size of the search space:

$$\prod_{f \in F} |f|$$

In the binary tree example, all fields generated by Korat have the same type (`Node`) and are assigned the same field domain. Therefore, all fields have the same number of possible values: $n + 1$ (n nodes and 1 null value). Each node has two fields (`left` and `right`) and there is one field in the `BinaryTree` itself, hence when using n nodes, there are $2n + 1$ fields to which Korat can assign one of the $n + 1$ `Node` objects. This leads to the following formula for the search space size of our binary tree example:

$$p(n) = (n+1)^{2n+1}$$

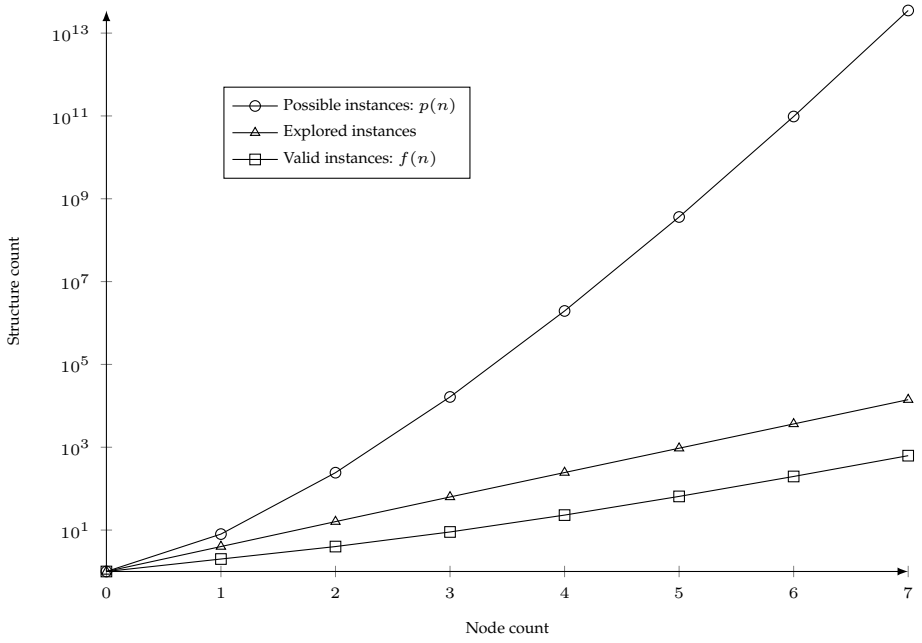


Figure 2.3: Generated binary tree structures in respect to number of nodes.

2.2.1.3 Putting it together

Figure 2.3 shows a plot of $p(n)$ and $f(n)$. In addition, there is also a plot for the number of instances Korat passes to the `repOK` method (labeled "Explored instances"). These values were obtained experimentally, by running Korat for the `BinaryTree` with the respective number of nodes. In addition it is worth noting that the number of valid instances (calculated by $f(n)$) matches the number Korat actually finds (the ones accepted by `repOK` and passed to the user's test method).

The plot depicts the number of structures for the respective category on the y axis in respect to the number of nodes on the x axis. It is a semi-logarithmic plot, meaning the y axis has a logarithmic scale, while the x axis has a linear scale.

Considering the function's complexities, it was to be expected that there are a lot more possible instances than there are valid ones. It is interesting to see that Korat is able to trim a huge portion of the search space (consider the logarithmic scale) and explore only a little more than what is actually valid.

As an example: With 6 nodes, the cardinality of search space is about 20 million larger than the number of valid instances. But Korat only explores (calls `repOK` for) 20 times as many objects as there are valid ones. Assuming a 3 GHz processor could generate and validate an instance within a single cycle, it would still take an hour to explore the entire search space.

2.2.2 Naive Algorithm

The naive algorithm simply iterates through the search space element by element. The validity of every element is evaluated by invoking the `repOK` method and depending on the result, it is passed to the user's test cases. This algorithm forms the foundation for iterating the search space, but it is only used in combination with the pruning and isomorphism optimization in Korat. Without the filtering optimizations, the runtime of the algorithm would be too slow for most applications as explained earlier.

2.2.2.1 Search Space Representation

The search space is explored using a so called candidate vector. It is like an iterator [18] for the search space structure: It identifies an element of the search space and can be incremented to move to the next element. This allows to write a very simple high level algorithm as shown in alg. 1 for exploring the search space: Using the range based for loop, the candidate vector is iterated (line 1). Each element which is accepted by the `repOK` method (line 2) is passed to the test case (line 3).

```
   Input: testCase
   Input: candidateVector
1  foreach candidate in candidateVector do
2  |  if executeRepOK(candidate) then
3  |  |  runTestCases(candidate)
4  end
```

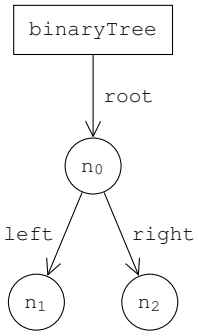
Algorithm 1: Pseudo code for state space iteration without filters.

The candidate vector is represented as a Java array of integers[4] with one element for each field specified in the `finitization`. Each element is an index into the field domain of the corresponding field. As described in section 2.1, a field domain consists of all possible values for its field. These values have a fixed order within the field domain and can therefore be referenced via an index. Hence, the candidate vector assigns a value to each field.

In the running example for the binary tree with three nodes, there is only one field domain, which is used for the node references. This field domain consists of two class domains. The first one was explicitly created and contains the three node objects. The second one was implicitly created and contains the `null` value we requested. Together, the two class domains form a field domain, which contains four values and has the following order: `null`, n_0, n_1, n_2. A candidate vector could for example reference n_1 using the index 2.

[4] The primitive integer type "`int`", as arrays may consist of primitive type and no null values are required.

Field	Index	Value
bT.root	1	n_0
n_0.left	2	n_1
n_0.right	3	n_2
n_1.left	0	null
n_1.right	0	null
n_2.left	0	null
n_2.right	0	null



Field	Index	Value
bT.root	1	n_0
n_0.left	2	n_1
n_0.right	2	n_1
n_1.left	0	null
n_1.right	0	null
n_2.left	0	null
n_2.right	0	null

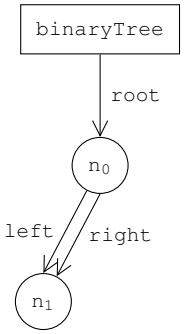


Figure 2.4: A candidate vector for a valid and an invalid tree.

Figure 2.4 shows two concrete candidate vectors and their corresponding tree object for the binary tree example. The table on the left of each tree represents the candidate vector. Each row of the table shows one element of the candidate vector. The first column (*Field*) identifies the field which is represented by this row. The second column (*Index*) is the index into the field domain and the third column (*Value*) shows the value to which this index corresponds.

The top part of the figure shows a valid binary tree with respect to the `repOK` method: There are no circles and each node is only used once. The tree in the lower part of the figure is, however, broken. The node `1` is used as a left and right child of the node `0`, which violates the invariant specified by the `repOK` method.

2.2.2.2 Search Space Iteration

In order to iterate through the search space, Korat needs to be able to increment the candidate vector. The algorithm to this is similar to incrementing a number where the base of each digit is the size of the corresponding field domain.

In the beginning, each element of the candidate vector is set to zero. For one increment Korat first increments the index of the last element. If the new index exceeds the field domain, it is reset to zero and the next element of the search space is also incremented, using the same overflow handling system. The search space exploration is completed as soon as the index contained in the first element overflows.

2.2.3 Pruning Filter

The idea of the pruning filter is that the result of the `repOK` method only depends on the fields it accessed. Therefore, it is independent of all fields it did not access. Hence, Korat knows that different values for the independent fields will not change the result of the `repOK` method.

2.2.3.1 Algorithm

Algorithm 2 shows the pseudo code for the pruning algorithm. The details of the algorithm are explained throughout this section. Whenever a field of the generated test structure is accessed, its candidate vector index is added to the accessedFields list if it was not already contained in it. In order to do this in the concrete implementation, Korat instruments the Java byte code as described in section 2.1.3).

After initialization, the algorithm starts by executing `repOK` (line 4). Each time the method returns `false` and thus marking the current candidate as invalid, Korat skips over all elements with different values for the independent fields. This is done by tracking which fields of the generated test structure are read during the execution of `repOK`. Only these fields can change the outcome of `repOK`.

As an example consider the invalid binary tree in fig. 2.4: After the execution of `repOK`, the list with accessed field indexes contains: `{0, 1, 2}`. This corresponds to

```
1  candidateVector ⟵ ⟨0, 0, . . . , 0⟩
2  do
3      accessedFields ⟵ ⟨⟩
4      if executeRepOK(candidateVector.candidate) then
5          runTestCases(candidateVector.candidate)
6          touchReachableFields(candidateVector.candidate)
7  while candidateVector has next
```

Algorithm 2: Pseudo code of pruning algorithm.

the fields: {bT.root, n_0.left, n_0.right}. Korat knows that changing other fields would not change the outcome of repOK. Therefore, all candidate vectors of the format {1,2,2,*,*,*,*} represent invalid objects, too. These $4^4 = 256$ candidates can be safely pruned from the search space.

In order to skip over these irrelevant fields, an algorithm similar to the naive algorithm is used. Instead of always incrementing the last element in the candidate vector, Korat starts by incrementing the element identified by the last entry in the list with accessed fields. If the incremented index exceeds the size of the field domain, Korat resets it to zero and moves on to the element in the candidate vector identified by the next entry in the list with accessed fields. Once there are no more entries in the list with accessed fields, the search is complete.

In our example, Korat first increments the third element in the candidate vector: n_0.right. After that, the next candidate is found, because the new index (3) does not exceed the field domain size (4). This new candidate represents the valid tree shown in fig. 2.4.

After that, the algorithm jumps back to the start of the exploration loop (line 3). It removes all entries from the accessedFields list and continues by executing repOK again. Each time the method returns true and thus marking the current candidate as valid, Korat passes the found object to the user's test cases (line 5). After that, Korat needs to make sure to iterate through all elements of the search space which have the same values for the fields and which were accessed during the execution repOK. All of these represent valid candidates, too. In order to do that, Korat adds all fields which are reachable starting from the root object (the one which is passed to the user's test) to the accessedFields list. Doing so, Korat is able to prune unreachable parts of the search space, as these parts can never be accessed by the user's test cases.

Assuming that the order in which the fields are accessed in the repOK method is deterministic, the pruning algorithm will always explore all valid elements of the search space. Otherwise, for non-deterministic repOK methods, the algorithm only guarantees that no invalid element is passed to the user's test cases. However, it is possible that the algorithm will miss valid elements, hence, the exhaustive exploration of the search space is no longer

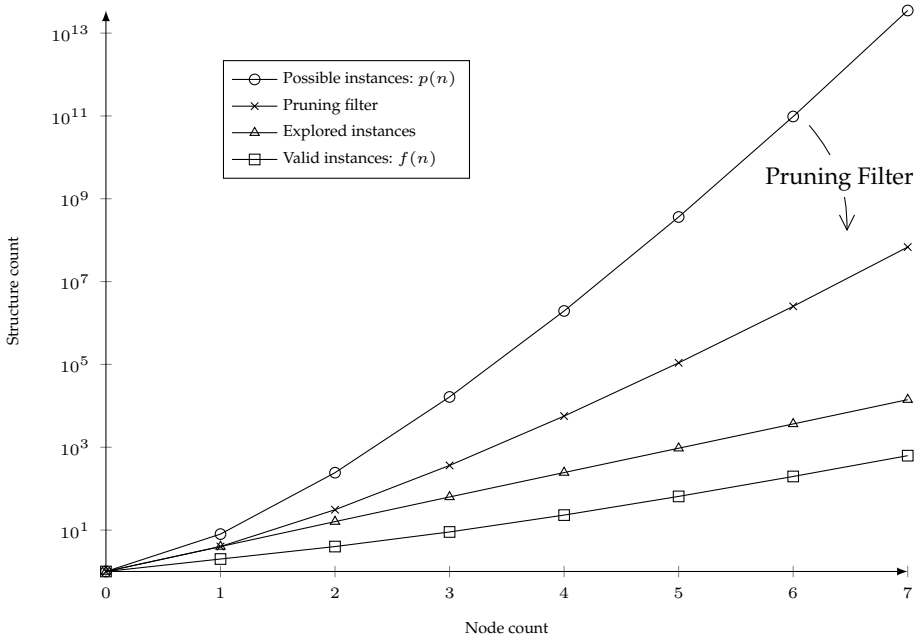


Figure 2.5: Generated binary tree structures in respect to number of nodes.

guaranteed. By disabling the pruning optimization, Korat could be used to explore the entire search space in a non-deterministic environment. As invariants (`repOK` methods) tend to be deterministic, this seems more like a corner case problem. Therefore, this thesis continues work under the assumption of a deterministic `repOK` method.

2.2.3.2 Effectiveness of Pruning in Practice

Figure 2.5 shows the benefit of the pruning filter. The number of nodes in the previous graph is depicted on the x axis and the number of generated structures on the y axis. Starting from the top, the first line shows the number of possible instances. The next line is new and it shows how this number is improved when using the pruning optimization. After that, the following two lines show the number of explored instances (passed to `repOK`) and the number of valid instances (passed to the user's test cases) again.

It is clear to see that there is already a huge improvement: for 7 nodes, the pruning filter reduces the number of generated instances from $3.5e13$ node to $6.8e7$ nodes.

In order to achieve efficient pruning, the `repOK` method should reject invalid instances as soon as possible. The lesser fields it touched before returning `false`, the better Korat is able to prune the search space. The worst case for the pruning optimization is a `repOK` method which always touches all fields of the structure before deciding if it is valid. This prevents Korat from pruning anything. Korat has to assume that the result of `repOK` depends on all fields. Hence, all instance have to be explored (passed to `repOK`).

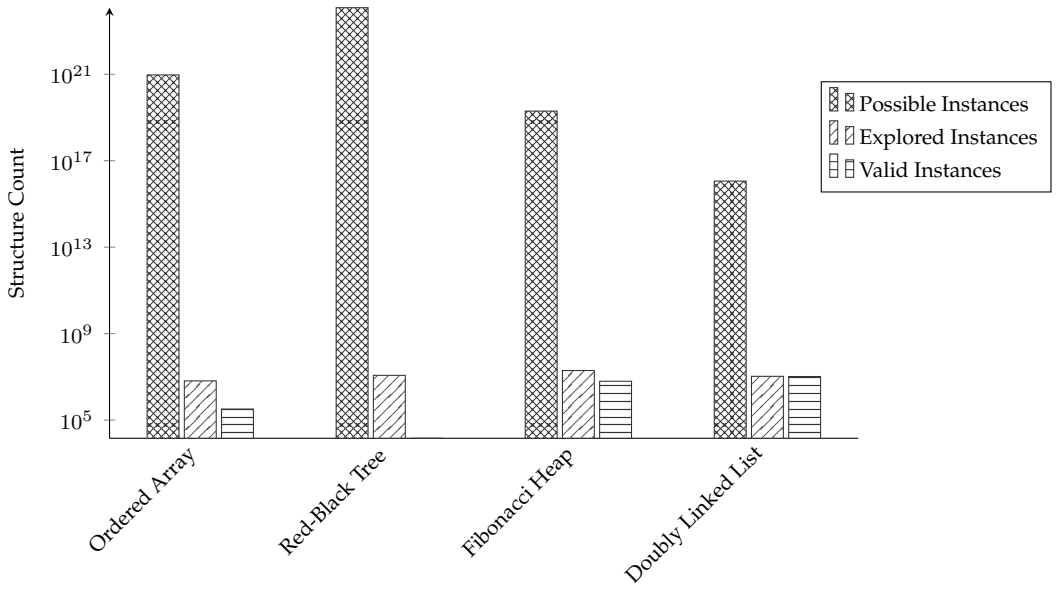


Figure 2.6: Pruning filter effectiveness for different data-structures.

As stated by the original Korat authors [12], the algorithm works well in practice:

> "In practice, our search algorithm prunes large portions of the search space, and thus enables Korat to explore very large state spaces."

The experiments shown in the paper show how much of the search space is explored when using both the pruning and the isomorphism filter. We conducted an experiment to see the effectiveness of the pruning filter in isolation. Its results are shown in fig. 2.6. The y axis shows the number generated structures. Each group of three bars on the x axis shows one of the tested data-structure:

- **Ordered Array**
  An array of size 30. Each element is an integer between one and five. Only the ones sorted in ascending order are considered valid.
- **Red-Black Tree**
  Simple red-black tree implementation where each node has a parent reference, two child references, a key and a color. Six nodes were used in the experiment.
- **Fibonacci Heap**
  Each of the five nodes of the Fibonacci heap [15] used in the experiment keeps a reference to its parent, right, left and child node. In addition each node is assigned its degree and a key.

- **Doubly Linked List**
  An implementation of a doubly linked list: Each element has a reference of its predecessor and its successor. `Null` marks the end and the start of the list. In addition each list element holds a value object.

The diagram shows that the pruning filter is able to achieve a significant reduction for the number of explored instances for every tested data-structure. It reduces the number of explored instances by a factor of 10^{10} to 10^{15}.

2.2.4 Isomorphism Filter

The benefit of the isomorphism filter is twofold: First, it decreases the number of explored instances (the ones which are passed to the user's test method) and thus improving the performance. Second, it prevents calling the user's test method with structurally equal object graphs, i.e. isomorphic ones.

In order to explain the algorithm, we first describe what it means for two object graphs to be isomorphic in section 2.2.4.1. After that, the algorithm used by Korat for the filter is explained in section 2.2.4.2. A different and more detailed explanation including a prove for the correctness of the algorithm can be found in [33]. Lastly, in section 2.2.4.3, we show how this filter helps to decrease the number of explored instances in practice and thereby allows the tester to explore larger search spaces.

2.2.4.1 Isomorphism Definition

In object oriented languages like Java, there are usually two kinds of variables, one of which holds a reference to an object and the other one stores a primitive value. For storing a primitive value, the physical memory of the variable simply contains this value. For storing a reference, the physical memory contains the identity of the other object[5]. This identity can be used to find the other object and use it. Hence, each object has a unique identity. In Java, this identifier can be obtained using the `System.identityHashCode` function.

This implies that two different objects are never equal, because their id always differs, though the structure they represent can be equal. This is basically the idea of the isomorphism in Korat: When generating structures, Korat ignores the object's identity[6]. Hence, when comparing two object graphs, only the actual values matter and not the ids of the involved objects. In Java, this is also the intended difference between comparing two objects via the == operator and using the `equals` method: The == operator simply checks if the identity of the operands is the same. The `equals` method checks whether the represented structure is equal[7].

[5] Usually the identity of an object is its physical address.

[6] Korat supplies an option to disable this mechanic.

[7] The `equals` method needs to be implemented by the user and therefore can do anything, but the described behavior is the usual and intended one.

```
   Input: o1, o2          // Objects to be compared
1  if o1 = null or o2 = null then return o1 = o2
2  if typeOf(o1) ≠ typeOf(o2) then return false
3  foreach field in o1 do
4      if field is primitive then
5          if o1.field != o2.field then return false
6      else
7          if (o1.field, o2.field) ∈ visited then continue
8          visited.add(⟨o1.field, o2.field⟩)
9          if not areIsomorph(o1.field, o2.field) then return false
10     end
11 end
12 return true
```

Algorithm 3: `areIsomorph` Method in pseudo code.

Algorithm 3 shows a recursive definition for Korat's isomorphism. A relational definition can be found in [12]. We chose this definition as it is more intuitive to read and therefore simpler to understand. The `areIsomorph` function gets two objects o_1 and o_2 as input. It returns true if the two objects are isomorph and false otherwise.

First, the algorithm has to do a `null` check: If either of the objects is null, the other one has to be null, too. Otherwise the algorithm aborts and returns false. If both input objects are not `null`, the algorithm continues in line 2. There, the type of the two objects is compared. If it is different, the structures are obviously not isomorph and the algorithms returns false.

Otherwise, for two objects with the same type, the structure is isomorph if all of their fields are isomorph. Hence, the algorithm loops over all fields (line 3). Primitive and reference fields are treated differently: For primitive fields (line 5), the value of the field has to be equal in both objects. For reference fields (line 7-9), only the referenced objects have to be isomorph to each other. To validate this, the algorithm recursively invokes itself if the object pair has not been visited before.

2.2.4.2 Algorithm

Using the isomorphism definition, one can partition the search space into partitions which only contain isomorphic object graphs: isomorphism partitions. The goal of the isomorphism filter is to only explore one object graph out of every partition. Korat traverses the search space in a fixed order[8]. The idea of the isomorphism filter is to only explore the first

[8]The order is implied by the list of access fields after each call to `repOK` together with the ordering in the field domains

element of every isomorphism partition in respect to the exploration order. Conceptually, this is achieved by incrementing the index in the candidate vector of a field by more than one and thereby skipping isomorph structures.

```
   Input: F
   Input: f
1  m_f ⟵ max({index_cd(f') | f' ∈ F ∧ cd(f') = cd(f)} ∪ {−1})
2  if index_cd(f) ≤ m_f then
3  |  index_fd(f) ⟵ index_fd(f) + 1
4  else
5  |  index_fd(f) ⟵ index_fd(f) + cd(f).size − index_cd(f)
6  end
7  if index_fd(f) ≤ fd(f).size then
8  |  found new candidate
9  else
10 |  continue backtracking
11 end
```

Algorithm 4: `skipIsomorphStructures` Method in pseudo code.

The details of the algorithm are shown in alg. 4 as pseudo code. The single steps of the algorithm are illustrated using the example in fig. 2.7, which shows one step of the state space exploration. Like in fig. 2.4, a binary tree with three nodes is used. On the left hand side, the entries of the candidate vector are shown in a table, while the other side depicts the corresponding tree structure. For simplicity, the values for the `left` and `right` child of node n_1 and n_2 are left out, as they are always `null`. The shown step is an example for a step where the isomorphism filter skips some isomorph structures. In this case, the algorithm skips the structure where the `right` child of n_0 is n_2, as it would be isomorph to its predecessor.

The algorithm for the isomorphism filter builds on the pruning algorithm which was introduced in section 2.2.3. Recall how the pruning algorithm only increments the field domain index in the candidate vector of the fields which were accessed during the execution. Instead of simply incrementing the last accessed field (and backtracking in case of an overflow), the isomorphism filter code as shown in alg. 4 is used instead. Whenever it detects an isomorph structure, it increments the field domain index by more than one.

The isomorphism filter algorithm gets two input arguments: First, the field f in the candidate vector, which should be changed next. And second, the remaining fields in the list of accessed fields F (excluding f). In the example in fig. 2.7, f is the third element in the candidate vector, n_0`.right` and F contains ⟨`bT.root, n0.left`⟩.

The algorithm starts by calculating m_f. To do this, it takes the highest value for the class

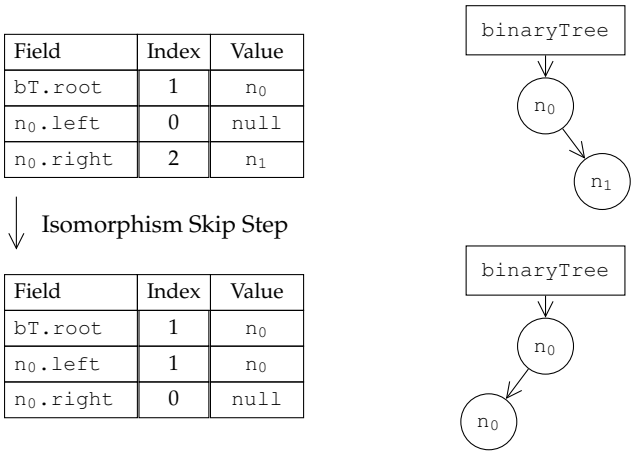


Figure 2.7: Isomorphism filter example with three nodes.

domain index[9] of all fields in F which refer to the same class domain as f does (if there are no such fields, -1 is used). The $\texttt{index}_{\texttt{cd}}$ function returns the class domain index of the field and `cd` function returns the current class domain of the field. In the example, the only field in F with the same class domain is `bT.root`, because $\texttt{m}_0\texttt{.left}$ refers to the `null` class domain. Therefore, m_f evaluates to 0.

Next, the algorithm tests whether incrementing the field domain index of f will generate an isomorph structure: If the class domain index of f is less than or equals to m_f, then the field domain index of f can be simply incremented. Otherwise the algorithm skips over all remaining elements of the current class domain. In the binary tree example, the else-branch is taken and the algorithm skips over all remaining objects in the `Node` class domain. Therefore, the field index of f is increased to 4 and points to the first element of the class domain after the node class domain, thus skipping the one structure where the `right` child of $\texttt{n}_0$ is $\texttt{n}_2$, because it would be isomorph to the previous one.

The original structure represented the smallest element of its isomorphism partition in terms of field indexes. Using an object from the same class domain with a larger index for the last accessed field would not change the structure, but only put a different object (in terms of object identity) in the same place. Therefore, Korat skips all remaining objects of the same class domain.

Lastly, the algorithm needs to check for an overflow of the field domain index of f. If so, the algorithm backtracks to the next field in the accessed fields list. Otherwise a new candidate was found and is passed to the `repOK` method. In the example, there was an overflow: The field domain of $\texttt{n}_0\texttt{.right}$ has only 4 elements (`null` and three nodes). Therefore, the field domain index for $\texttt{n}_0\texttt{.right}$ is reset to 0 and the algorithm is called

[9] Recall the difference between class and field domain: The class domain index is the index in the current class domain, while the file domain index is the index in the field domain. In the example, the field domain index of `bT.root` is 1 and its class domain index is 0.

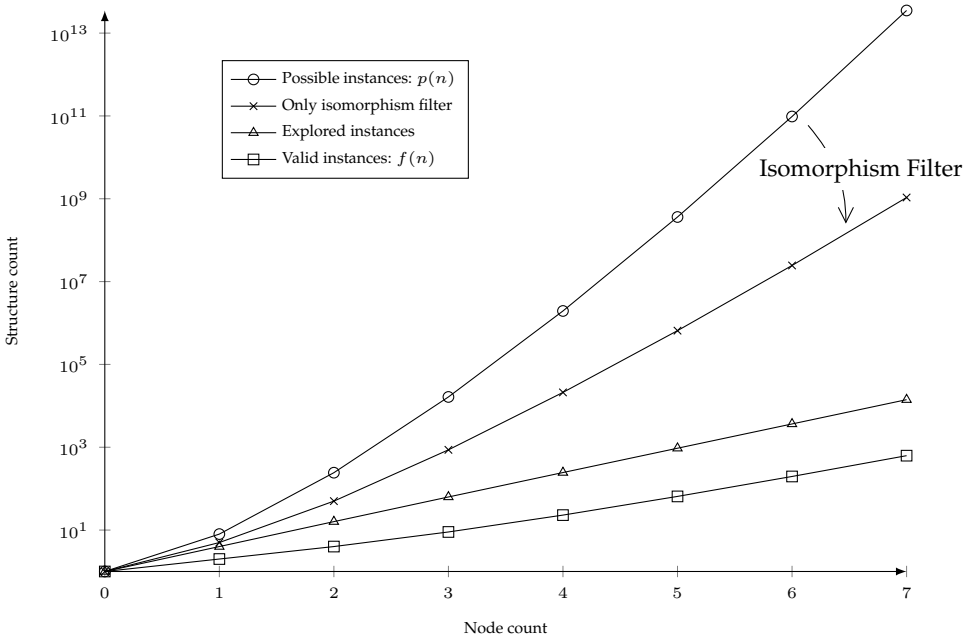


Figure 2.8: Generated binary tree structures in respect to number of nodes.

again for the next field in the accessed fields list: n_0.left. This time, the then-branch in line 3 is taken and the field domain index for n_0.left is incremented by one. The resulting structure[10] is shown at the bottom of fig. 2.7.

2.2.4.3 Effectiveness of Isomorphism Filter in Practice

Figure 2.8 shows the effectiveness of the isomorphism filter. As before, the number generated binary tree structures is plotted on the logarithmic y axis in respect to the number of nodes on the x axis. There are four lines in the diagram: The first one shows the number of possible instances. These are all object graphs Korat could possibly create using the given set of objects. The next one shows how many structures are created and passed to the repOK method using only the isomorphism filter. The third and fourth line show, just like in the previous plots, the number of explored and valid instances. The latter one depicts the instances which actually represent valid binary trees and which are passed to the user's test method. The line for the explored instances shows the number of object graphs being passed to the repOK method: the invalid ones are still included here.

The diagram shows that the isomorphism filter alone cuts off a large portion of the search space. For 7 nodes, only 1.1e9 structures are explored instead of 3.5e13. Therefore, this kind of filtering is less effective than the pruning filter, where the number of explored

[10]Note that the generated structure represents an invalid tree and will be filtered by the repOk method.

structures was reduced to $6.8e6$ for 7 nodes. But unlike the pruning filter, the isomorphism filter can only remove isomorph valid structures.

2.3 Modifications

The initial publications around Korat [12, 33] use the framework only for testing data-structures in isolation. Later studies [34] did not change this focus and kept Korat as a tool for testing data-structures. Korat proved itself very useful for automated test input generation for data-structures and therefore as a useful tool for for these specific projects, where custom data-structures are required.

This thesis aims to extend Korat to be applicable for testing larger applications. But by using Korat for hole projects instead of single data-structures, it also has to deal with a larger code base. While the number of lines itself is not a problem, the Java language features being used in larger applications is a problem, as some of them cause Korat to break.

The initial intend when starting to use Korat for testing the example game *Chelone* was to experiment with automated input data generation. But some of the issues described in this section made it impossible to use Korat for testing the *Chelone* code base, without rewriting large portions of it. Therefore, the authors took the liberty to resolve these issues in Korat first. This makes it possible to use Korat in a lot more projects and benefit from its automated test input generation.

This section lists the issues in Korat which were found throughout the research conducted in this thesis and documents the modifications. Korat's source code was released under the GNU General Public License 2 [3]. This allows programmers to use and modify the code freely as long as they also publish their modifications. In particular, this allows us to explore the code to fix problems in it.

In our efforts to apply Korat to the Chelone code base, which is written in Java 1.7, we ran into several issues as mentioned above. In almost all cases, Korat had problems with various code patterns in our code. One of the key contributions of this thesis are the modifications to Korat, which makes it work with more common code patterns in Java and thus improving its usability for the community.

2.3.1 Issue 1 - Reference to TestCradle

As described in section 2.2, Korat needs to track accesses to fields of the Objects it has generated. Therefore, each time a field is accessed, the central test object, named *TestCradle*, is notified via a method call. The TestCradle stores which fields were accessed during the execution of *repOK* and uses this information to generate the next test input.

In order to get these notifications, Korat modifies the byte-code with the help of the javassist[14] library. For each field of each class Korat generates a special getter method and wraps all read access behind this new getter method. Inside the getter method a notification method is called on the *TestCradle* object before the value of the field is returned.

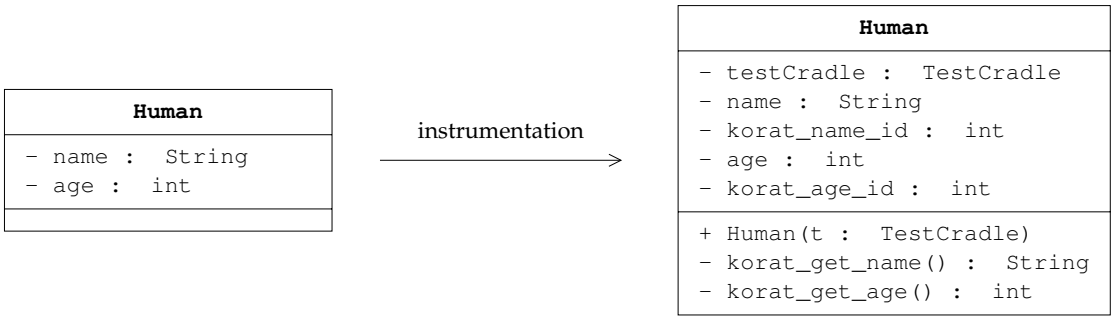


Figure 2.9: Initial State of the instrumentation.

To find out which field triggered a notification, Korat associates each field with a unique id. To achieve this, Korat further changes the byte-code and adds an id field for each field. The id field is initialized when Korat creates a new object for the test input. When the field is accessed via its special getter this id is passed via the notification method to the *TestCradle*, which is then able to determine which field is responsible for the notification.

The last thing missing in this system – and the reason for this issue – is a way to access the *TestCradle* object from within the generated getter methods. The following paragraphs will explain several solutions and their limitations, starting with the initial implementation used in the original Korat release and ending with the one this work came up with after several attempts.

2.3.1.1 Initial state

The instrumentation of the byte-code is done when a class is being loaded. The JVM loads a class only when it is being used (for example when it's being instantiated). This lazy class loading concept allows for a faster startup of the JVM and prevents wasting time on loading classes which are never going to be used. Korat hijacks this system by replacing the default class loader with a custom one: the so called *InstrumentationClassLoader*.

The *InstrumentationClassLoader* adds an id field and a special getter for each field. It also modifies the read accesses to call the special getters instead of reading the field directly. In the initial version of Korat it also adds a field for the *TestCradle* and a special constructor which gets a reference to the *TestCradle* as an argument and initializes the newly added field. Whenever Korat creates an object for a test case, it calls this special constructor and the reference to the *TestCradle* would be initialized.

This system is illustrated by the UML Class Diagram in Figure 2.9. On the left hand side it shows the original class `Human` with a `name` and an `age` field. On the other side it shows the transformed class after the instrumentation. An id field and a getter was added for each field. In addition, there is a new constructor and a field for the *TestCradle*.

2.3.1.2 Fix 1: Virtual Methods

This system works well until the code uses inheritance. When Korat creates the special constructor's body, it adds a call to the super classe's special constructor, if the super class was already instrumented. Otherwise it will only initialize the reference to the *TestCradle* in the current class. Hence, the *TestCradle* reference in the super class is not initialized.

As an example, consider two classes: `Derived` and `Base`, where `Derived` extends `Base`. We take a look at both instrumentation orders:

- When `Base` is instrumented first, a call to the special constructor of `Base` is added during the instrumentation of `Derived`. In this case, the *TestCradle* field in `Base` is initialized during the construction of the `Derived` class. Therefore, all accesses to the fields in `Base` are tracked correctly, because the *TestCradle* reference in `Base` is initialized.
- In the other case, when `Derived` is instrumented first, Korat does not add a call to the special constructor of `Base`. Therefore, the reference to the *TestCradle* in `Base` is not initialized. This has the consequence that Korat can not track the accesses to fields in `Base` and assumes that nothing was accessed during the execution of `repOK`. This has the consequence that Korat may prune portions of the search space, which could potentially contain valid input configurations.

In our first attempt to fix this problem, we created a getter method for the *TestCradle* field in each class we instrument. Whenever we needed to access the *TestCradle*, we'd call the getter method. This method call is dispatched by the JVM at runtime and always executes the implementation of the most derived class. Therefore, the method is always executed in the class of the object itself. The *TestCradle* field of this class is initialized correctly, as we called the constructor of this class when we created it. Therefore, we can always find a valid reference to the *TestCradle* and notify it correctly.

2.3.1.3 Fix 2: Global Reference

This initial fix works well until enums are used. Enumerations or enum types are rather powerful in Java. Just like in other languages, an enum type can only have a fixed set of instances, called enum constants. But unlike in many other languages, these enum constants can define functions, constructors and fields. Without going into too much detail, one interesting example for using these enum features is shown in listing 2.4. It was taken from the Java specification [26], to which the interested reader is referred to for more details on enums.

From the perspective of Korat, it is only important that enums are able to contain fields, because these need to be tracked. The problem is the reference to the *TestCradle*: Each enum constant is created at class load time by the JVM itself and it is prohibited by the Java language specification to create new enum constants. Hence, we are not able to call our constructor and initialize the *TestCradle* reference.

```
enum Coin {
   PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

   Coin(int value) {
      this.value = value;
   }

   private final int value;
   public int getValue() { return value; }
}
```

Listing 2.4: Capabilities of enum types in Java.

The easiest way to solve a dependency issue like this is to introduce a global variable which references the *TestCradle*. Each time a field is accessed, the special getter uses the global variable to obtain a reference to the *TestCradle* and calls the notification method.

2.3.1.4 Fix 3: Injection

Global variables are considered bad practice in modern software design and Korat already uses several global variables, which have caused problems in other parts of the system (see section 2.3.4). Introducing more dependencies on these global variables would likely lead to more problems in the future. Therefore, a better solution was developed:

When Korat initializes a search space, it creates each object which was defined in the `Finitization` upfront (enums are directly passed in the `Finitization` and not created). Previously this was done using the special constructor, which received the *TestCradle* object as an argument. Now, the default constructor is used and Korat simply sets the fields directly using reflection in the class itself and all its parent classes.

2.3.2 Issue 2 - Static Arrays

As explained in section 2.1.3, Korat monitors arrays by wrapping them inside objects and thereby tracking access to the array's entries. To make this work, Korat needs to replace each read and write access to the array by a method call to the wrapper object.

This technique was also used for static arrays. By doing so, Korat broke essentially all `switch` statements. The reason for this is that Java compilers often handle `switch` statements by introducing a static array, which serves as a jump table. If Korat instruments the class and replaces the array by a wrapper object, the switch statement does not work anymore and the program crashes at runtime.

This issue was fixed by removing the instrumentation for static arrays. This is a valid fix, because Korat does not support the generation of static variables anyway. In addition, the usage of static variables is not a very good object oriented design pattern.

2.3.3 Issue 3 - Non-Static Arrays

During our experiments, we also encountered a few low level bugs in the Java byte code. These issues were caused by the instrumentation of the byte code. The solution for these bugs is usually rather simple. The difficult part is finding them. In order to not exceed the scope of this thesis, only one of these bugs is described here to give an idea of the type of problems which can arise when instrumenting large amounts of source code.

2.3.3.1 Normal Instrumentation

In order to track accesses to an array, Korat replaces it with a wrapper object, a so-called KoratArray. Whenever an entry of the original array is read or written to, the access is replaced by a method call on the KoratArray. This is achieved by altering the byte code. Assume a read from an `int` array:

```
public void test () {
   int var = myIntArray[2];
}
```

Using a standard Java compiler, the following byte code is generated:

```
ALOAD_0
GETFIELD
ICONST_2
IALOAD
ISTORE_1
```

First, `ALOAD_0` loads the `this` reference onto the stack. The `GETFIELD` instruction is used to load a field from the object on top of the stack. In this case, it loads the `myIntArray` member of `this` onto the stack. Next, `ICONST_2` puts the constant 2 onto the stack. The following instruction consumes the two values which were loaded onto the stack. It is used to load an element from an array. In this case it loads the second element of the `myIntArray` array onto the stack. Lastly, this value is popped from the stack and stored in the first register using `ISTORE_1`.

Korat needs to replace the access to the array – namely `IALOAD` – by a method call to the wrapper object. This is done by replacing `IALOAD` with a `INVOKEVIRTUAL` instruction. The new instruction works on an object reference and an integer value, instead of an array reference and an integer value. In the instrumented code, Korat has replaced the `myIntArray` array by a wrapper object. Therefore the `GETFIELD` instruction puts an object reference onto the stack instead of an array reference. Hence, the `INVOKEVIRTUAL` instruction gets the correct input types.

The `IALOAD` instruction is one byte in size, while the `INVOKEVIRTUAL` instruction requires three bytes: One byte for the instruction itself and two byte for the method descriptor. Because of the larger size of the new instruction, Korat can not simply replace it. To work around this problem, the javassist library offers a function to insert a gap into the

Line	Original Code	With Gap	After Overriding
0x0	IALOAD	NOP	INVOKEVIRTUAL
0x1		NOP	
0x2		NOP	
0x3		NOP	NOP
0x4		IALOAD	IALOAD

Figure 2.10: Byte code instrumentation bug.

byte code. Using this function Korat can insert two byte after the `IALOAD` instruction and then override the three bytes with the `INVOKEVIRTUAL` function.

2.3.3.2 The Buggy Part

A problem may arises when there is a jump after the array access. The JVM does not allow to jump to any address in the byte code, therefore the jump target needs to be aligned. When javassist adds a gap of two bytes, it may have to add a larger gap instead in order to keep the alignment of the following code intact.

Figure 2.10 shows the problem which occurs. The first column shows the byte number and the second one the original byte code. It is one byte long and only contains the `IALOAD` instruction. Next, in column three, the byte code after the gap insertion is shown. The gap was inserted before the `IALOAD` instruction and instead of the requested two bytes, javassist inserted four bytes to keep the alignment constraints of the following code intact. In the next step Korat overrides the first three bytes with the `INVOKEVIRTUAL` instruction. The resulting byte code is shown in the fourth column.

Essentially Korat expected the inserted gap to always be 2 byte in size. If this is not the case, Korat no longer overrides the previous instruction. This causes a crash during the execution, because two additional elements are popped from the stack and thereby corrupting the stack.

Once one understands the problem, there are a few straightforward ways to solve it. The interesting part is finding out what's going on. One possible solution is to insert the gap after the `IALOAD` instruction or Korat could override the `IALOAD` instruction with a `NOP` instruction before inserting the gap.

2.3.4 Issue 4 - Global Variables

Global variables are evil and should be avoided whenever possible.

- On a design level, global variables encourage coupling by making dependencies between modules possible and easy to create, which can lead to a bad architecture. A global variable exposes internal information of a module to all other parts of the

system. As soon as one other part or even a customer starts using it, it becomes extremely difficult to change or remove it.

- On a function level, global variables make it difficult to read code, as they have an infinite scope, which makes reasoning about their state extremely difficult. The scope of a variable should be as small as possible. If a variable is only being used in a small confined scope, it is easy to reason about its state. The larger the scope of the variable gets, the harder it is to understand what is going on with the variable, because it is used from many different contexts and each one has to be understood first.
- On a application level, global variables make it difficult to run the code more than once without restarting the program. Unlike member variables they are not initialized or reset when the object containing them is created. A method which only uses non-static variables can be run multiple times without any problems. If global variables are involved, this becomes difficult, as they need to be explicitly reset. In addition, running the method several times in parallel can be even more difficult.

These points have been discussed many times and are certainly not the focus of this work. Though, since we ran into problems with each of the mentioned points, we wanted to state this issue again: Korat uses a few global variables and they made it unnecessarily difficult to understand the code. We had to add code for resetting them, to be able to run Korat multiple times within a JUnit test suite, and because of the dependencies to the global variables, some refactoring tasks became needlessly complicated.

2.3.5 Issue 5 - Array Wrapping Rewrite

This issue, as some of the previous ones, is also related to arrays. But instead of bugs in the instrumentation, this problem is of a different nature: As already stated, Korat replaces every non-static array by a so-called KoratArray. These objects wrap the original array and notify Korat whenever an entry of the array is read. To make this work, all accesses to the original array are replaced by respective method calls to the wrapper array.

This works well for operations on the entries of the array: read or write the nth entry. A problem occurs when the array itself is read or written to, e.g. by assigning it to a variable, calling a function with it or returning it. The original implementation of Korat did not handle many of these cases well, as they rarely arise in single data-structure scenarios. We will first show two representative examples for places where this causes problems and then outline our solution to the issue.

2.3.5.1 Assignment to Local Variables

Consider the following code snippet:

```
1 public void test () {
2    int[] local = member;
```

```
3     int l = local.length;
4 }
```

Assume that `member` is a non-static field of the class of the shown method. Its type is `int[]` and its value is generated by Korat. Within the shown method, it is assigned to the local variable `local` and then the length of `local` is acquired.

In the initial Korat release this code raises an exception, saying:

> "java.lang.VerifyError: Bad type on operand stack in arraylength"

The problem is that the JVM tries to get the length of an array in line 3, while `local` references a KoratArray in actuality. Therefore, an object lies on the top of the stack, while the instruction for getting the length of an array (`ARRAYLENGTH`) expects a reference to an array. This type miss match is detected by the JVM and the program is aborted.

For arrays in member variables Korat replaces the calls to their length field by a method call to the respective KoratArray method. This technique could also be used for local variables. The problem is that Korat can not know if the local variable contains a KoratArray: The local variable could have also been assigned from a static array or it could have been returned from another method.

2.3.5.2 As a Return Value

A similar problem arises when using the following code:

```
1 public int[] test() {
2     return member;
3 }
```

As in the previous code snippet, assume that the variable `member` is an `int[]` array and its value is generated by Korat. The encapsulating class of `member` offers a public getter function for accessing it: `test`. Within the function the array is simply returned. When executing the code, the JVM detects a type error: "java.lang.VerifyError: Bad return type". The method is expected to return an array of type `int[]`. But instead, a KoratArray is returned. If Korat would change the return type of the method, all code which uses the method and is not instrumented by Korat would break.

2.3.5.3 Solution: Wrapping and Unwrapping

Next to the two described problems, the replacement of plain arrays by KoratArrays also causes other similar problems in a number of places. Therefore, when testing code with arrays and one of these problems occurs, the developer has to change his code or create the test input manually without the help of Korat.

To solve this bug, there are many possible solutions. The one we chose is intentionally rather simple: Whenever the KoratArray is read, its internal array is returned and all entries are marked as read. This keeps the scope of the KoratArray objects as little as possible

and thereby prevents the described problems from occurring. In addition, this solution has the advantage of being similar to the original one. Furthermore, it is rather easy to test that this implementation works for all possible usages[11] of an array. By immediately unwrapping the KoratArray, the remaining code remains unchanged and only operates on plain Java arrays.

The disadvantage of this approach is that it introduces false reads: As an example, assume that Korat generates an array with two entries. If the `repOK` method only accesses the first entry and then returns `false`, Korat knows that the result is independent of the second entry. Using the pruning filter, as introduced in section 2.2.3, this portion of the search space does not need to be explored. A false read occurs, when the `repOK` method first stores the array in a local variable. In this case our new implementation marks all entries of the array as read and thereby prevents Korat from pruning the search space.

Note that the false reads do not change the correctness of Korats state space exploration. They only slow it down, as larger portions of the search space can no longer be pruned. While this is certainly not a perfect solution, it is preferable to crashing.

To implement this approach, two functions are added to the KoratArray class:

- **`unwrap`**
  The `unwrap` method is called whenever an array field is read. This happens when the array is assigned to a local variable, to a static variable or to another field. Also, passing an array as an argument to a method or returning it from a method also requires reading the array field.

  Whenever a read on the KoratArray is required, the new solution calls the `unwrap` method on the KoratArray. It notifies the Korat that each entry of the array was accessed and then returns the internal array.

  On a byte code level, the normal read operation is performed by a `GETFIELD` instruction. In order to invoke the `unwrap` method, we simply add a `INVOKEVIRTUAL` instruction. This instruction calls a method on the object on top of the stack. After the `GETFIELD` instruction, this object is the KoratArray. The array is popped from the stack and by setting the method descriptor of the `INVOKEVIRTUAL` instruction appropriately (i.e. to `unwrap`), the `unwrap` method is called. When the unwrap method returns, it adds the internal array to the top of stack.

  When executed, the new byte code (with the additional `INVOKEVIRTUAL` instruction), replaces the KoratArray on top of the stack with its internal array. Hence, all following code can simply use the plain array.

- **`wrap`**
  This method is used whenever a KoratArray is written to. It takes a plain array as an input parameter and assigns it to the internal array of the KoratArray. In addition, it sets a flag which indicates that the internal array was overwritten. If the flag is set,

[11] The Korat regression test suite contains all cases where arrays are read and written two.

Original Code	Stack
PUTFIELD	$\langle objRef; array \rangle$
	$\langle\rangle$

Modified Code	Stack
SWAP	$\langle objRef; array \rangle$
GETFIELD	$\langle array; objRef \rangle$
SWAP	$\langle array; kArrayRef \rangle$
INVOKEVIRTUAL	$\langle kArrayRef; array \rangle$
	$\langle\rangle$

Figure 2.11: Instrumentation for calling wrap method.

no more reads to the array's entries are reported to Korat, because these new entries are not the original ones which were generated by Korat.

Instrumenting the Java byte code to call the `wrap` method is a bit more complicated than in case of the `unwrap` method. Figure 2.11 shows the original code and the modified code. Each line of both tables shows the byte code instructions on the left and the stack before the execution of the instruction on the right. The stack is visualized as an ordered sequence, where the top of the stack is shown on the right side[12].

On the left hand side of fig. 2.11 the original byte code for setting an array field is shown: The `PUTFIELD` instruction takes two arguments from the stack. It assigns `array` to the field described by its field descriptor in the object referenced by `objRef`. When the instruction has been executed, the stack is empty. Therefore the instrumentation also needs to produce code, which ends with an empty stack.

On the right hand side of fig. 2.11 the modified code with the call to the `wrap` method is shown. Recall that this code calls the `wrap` function on the KoratArray field of the object referenced by `objRef` instead of directly assigning a field in `objRef`. The `wrap` function then handles the assignment internally as described above. In order to call a function on the KoratArray, we first need to get a reference to the KoratArray onto the top of the stack. The `GETFIELD` instruction can be used to accomplish this. But it requires a reference to the object which contains the KoratArray: `objRef`. Therefore, we first have to use the `SWAP` instruction to swap the two top most elements on the stack. Now (line 2) the top of the stack contains the `objRef` and we can use the `GETFIELD` instruction to retrieve the KoratArray reference: `kArrayRef`. Next, we need to use the `INVOKEVIRTUAL` instruction in order to call the `wrap` method on the KoratArray. This method call, naturally, requires its arguments to be ordered. Hence we need to swap the two top most arguments of the stack again before we can call the `INVOKEVIRTUAL` instruction. The `wrap` function then handles the assignment and after it returns, the stack is clear, just like in the original byte code. Thus, the changes won't affect the following code.

[12] We only show the top most portion of the stack, which is relevant for the instrumentation.

These two functions handle the read and write to KoratArrays. It limits the scope in which the KoratArrays are used and thereby limits the required changes to the byte code. This makes it easier to test the instrumentation: The new test suite for Korat contains tests for each possible usage of a Korat array. This shows that KoratArrays can be used in more contexts now, which allows for a broader use of Korat.

2.3.6 Issue 5 - Finitization Rewrite

The finitization describes the search space by telling Korat which objects to use and how they are connected (see section 2.1.1). The implementation by the original authors has one shortcoming: There can only be one class domain for each class. The consequence is that two fields referencing the same class are always forced to use the same class domain.

Section 2.3.6.1 describes the original implementation in more detail and shows an example for potential problems. In the following section, section 2.3.6.2, we explain our solution to the problem.

2.3.6.1 Finitization as Bipartite Graph

For each field which is added to the finitization, a field domain is created. This field domain can contain any number of class domains and a class domain contains a set of objects of the associated class. As already mentioned, the initial implementation only allowed one class domain for each class. Therefore, there can only be one set of objects for each class. This restriction causes a problem when the same class is used in two different contexts.

As an example consider a class graph with two classes: `Person` and `City`. A `Person` has a `name` and a `residence` field. The first one contains a `String` object and the second one a `City` object. The `City` class also has a `name`, which is also modeled as a `String`. Using the finitization in fig. 2.12 we can create various persons using Korat.

The code first creates a finitization for the `Person` class, making it the root of our object graph. Then the `cityFD` is created with one `City` object. In order to do this, the `createObjSet` method creates a class domain with one `City` object and wraps it in a field domain. This field domain is then associated with the `city` field in the `Person` object in line 5. The following code block creates a `String` class domain, adds two sample names, wraps it in a field domain and associates this field domain with the two `String` fields: `Person.name` and `City.name`.

The described graph is shown in fig. 2.12. Rectangular boxes represent fields, while ellipse boxes show class domains. The problem is that we can only use one class domain for the `String` field[13]. Hence, it is not possible to have different sets of `String`s for the city names and the person names. Both fields reference the same class domain.

This issue occurs whenever one class is used in two different contexts. This is often the case with library classes like `String`. In addition, classes with generics, like collections,

[13] Korat's `createClassDomain` method uses a lookup table to check if the requested class domain already exists.

```
1  public static IFinitization finPerson(int unused) {
2  IFinitization f = FinitizationFactory.create(Person.class);
3
4  IObjSet cityFD = f.createObjSet(City.class, 1);
5  f.set("city", cityFD);
6
7  IClassDomain nameCD = f.createClassDomain(String.class);
8  nameCD.includeInIsomorphismCheck(false);
9  nameCD.addObject(new String[]{"Scott", "Tiger"});
10 IFieldDomain objSet = f.createObjSet(nameCD);
11 f.set("name", objSet);
12 f.set("City.name", objSet);
13
14 return f;
15 }
```

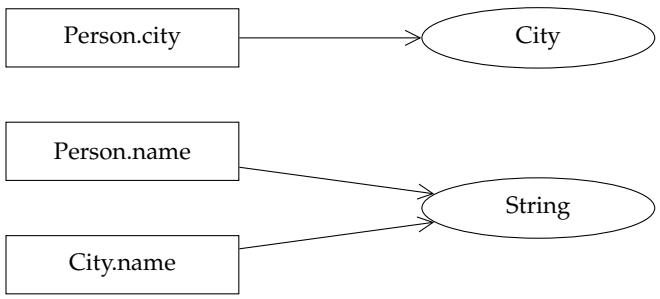


Figure 2.12: Old way of creating a tree like finitization.

can also not be distinguished.

2.3.6.2 Finitization as Tree

We solve this problem by rewriting the finitization and allowing the user to use more than one class domain per class. Doing so requires some changes in the finitization API. While refactoring the code we also made the interface more object oriented: Instead of calling all functions on the finitization object, it is now possible to call the function to assign a class domain to a field directly on the class domain object. This rewrite also lead to some modifications in the state space creation process[14], but as there are no semantic changes, they are not discussed here.

The new way of specifying a finitization is still pretty similar to the old way, making it easy for the user to adopt to the changes. The important changes are in the underlying structure. The code snippet in fig. 2.13 shows the code for creating a finitization for the previously introduced object graph with `Person` and `City`.

In the new code, it is possible to create two class domains for the `String` class. This allows us to specify two different set of objects for the two `name` fields. In addition, there are some extensions to the API, like the possibility to chain calls to class domains and the removal of the global `FinitizationFactory`. Notice the different ways of assigning a field domain to a field: It is no longer possible to assign a field domain to a field using the class name of the class in which the field is contained (line 11 in the old finitization code). The class domains are no longer unique per class, therefore the user has to explicitly specify the class domain in which the field should be set (line 10 and 15).

The resulting structure is also shown in fig. 2.13. Now there are two class domains for the `String` class. This allows the user to use a different set of names for the `Person`s and the `City`s. The new finitization also allows to create two class domains for objects with a generic type. Therefore, it is now possible to have two collections of the same type with different contents.

[14] The algorithm now has to traverse the finitization tree instead of looping over the flat representation of the old implementation

```
1  public static IClassDomain finBinaryTree(IFinitizationContext fc) {
2     IClassDomain personCD = fc.createClassDomain(BinaryTree.class, 1);
3
4     IClassDomain cityCD = fc.createClassDomain(City.class, 1);
5     personCD.set("residence", cityCD);
6
7     IClassDomain pNameCD = fc.createClassDomain(String.class, 0);
8     pNameCD.addObject("Tiger").addObject("Scott");
9     pNameCD.includeInIsomorphismCheck(false);
10    personCD.set("name", pNameCD);
11
12    IClassDomain cNameCD = fc.createClassDomain(String.class, 0);
13    pNameCD.addObject("Munich").addObject("Augsburg");
14    pNameCD.includeInIsomorphismCheck(false);
15    cityCD.set("name", cNameCD);
16
17    return personCD;
18 }
```

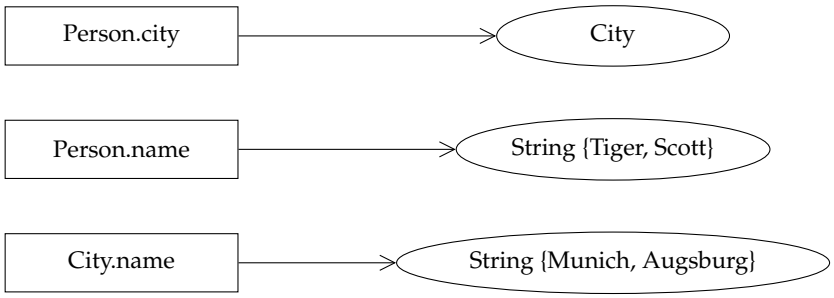


Figure 2.13: New way of creating a tree like finitization.

3 TestEra

TestEra [28, 35] is a specification based testing framework for Java code. More specifically, it is a tool for automated test case generation: Given a formal specification, TestEra is able to generate all non-isomorphic instances[1] of a given data-structure or object graph up to a specified size. These generated objects can then be used as inputs for user written test cases.

This chapter is laid out as follows: First, section 3.1 introduces TestEra from a user's perspective using the same binary tree as an illustrative example as in chapter 2. Next, section 3.2 takes a look at the inner workings of TestEra. The required algorithms are explained and the internal representation of the test structure is outlined. Lastly, an overview of TestEra's limitations is given in section 3.3.

3.1 Description

TestEra is based on Alloy [25] and the Alloy Analyzer [22, 24]. Alloy is a first-order relational language which is used to write the specification itself. The Alloy Analyzer is then used to create valid instances using this specification. TestEra connects these tools with Java by providing transformations between the Alloy and Java language.

We will first give a general overview of TestEra in section 3.1.1. After that, the workflow when using TestEra is explained in section 3.1.2. Lastly, section 3.1.3 gives an example, which shows how TestEra can be used to generate binary tree instances.

3.1.1 Overview

Figure 3.1 shows a high level overview of the process used within TestEra: The *TestEra Specification* is written by the user and describes the structure which should be generated: How many objects of which types are involved and what are the constraints for their connections and their primitive fields. The specification is written inside of Java annotations. It is expressed in the Alloy language with minor TestEra specific extensions.

TestEra extracts these annotations and combines them into a readable format for the *Alloy Analyzer*. Two Alloy files are extracted: First, an *Alloy Output Specification* and second, an *Alloy Input Specification*. The latter one consists in a so called Alloy Model, which defines several Alloy Signatures, which can be thought of as the counterpart of a Java class: a

[1] The authors claim that TestEra only produces non-isomorphic structures. But our experiments revealed a bug regarding this claim. For more details see section 3.3.2.

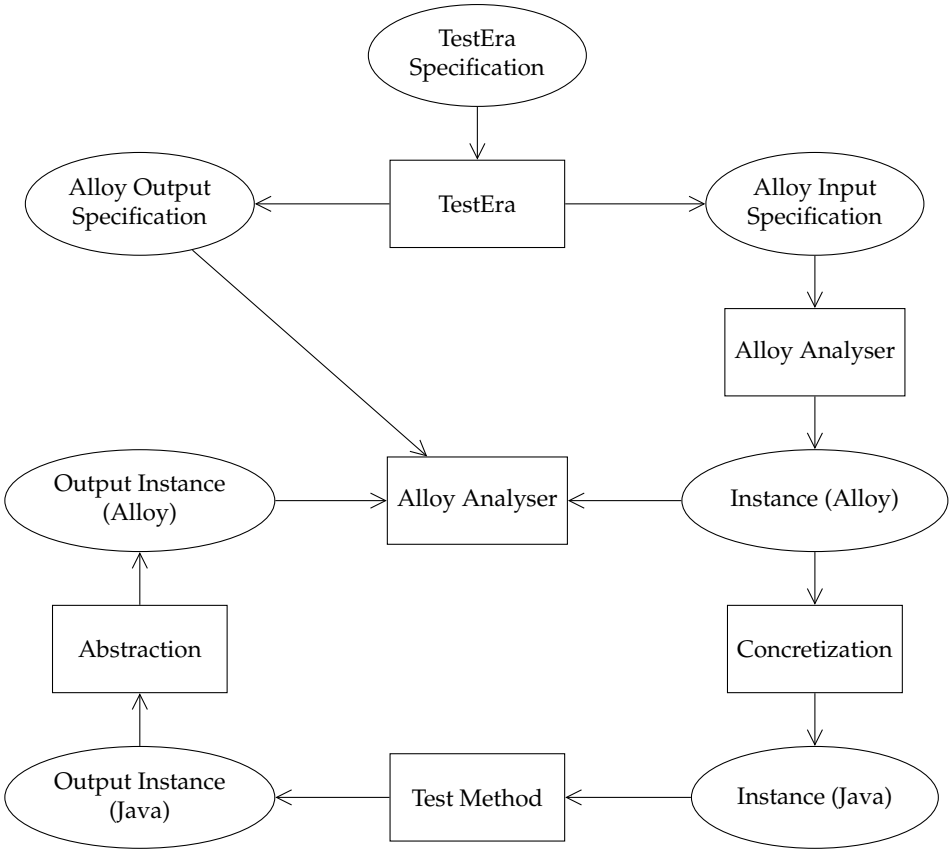


Figure 3.1: Overview of control flow in TestEra.

building plan for instances in the respective language. Unlike in plain Java, an Alloy Model also contains constraints on its instances. These are expressed as Alloy Facts and used by TestEra to encode the constraints specified by the user in the *TestEra Specification*. All together, the *Alloy Input Specification* describes the structure to be generated.

Given a bound on the number of instances (also contained in the *TestEra Specification*) for each Alloy Signature, the *Alloy Analyzer* is able to generate all valid non-isomorphic instances. TestEra automatically creates a *Concretization* to transform these instances (called "*Input Instance (Alloy)*" in the diagram) into Java instances. These can now be passed to the users *Test Function*.

In order to validate the results, TestEra offers the option to specify postconditions for the method under test. These are contained in the *Alloy Output Specification*, which is, as previously stated, extracted from the user's specification by *TestEra*. For the *Alloy Analyzer* to be able to verify these, an automatically generated *Abstraction* is used to transform the *Output Instance (Java)* back into an *Output Instance (Alloy)*. These are then verified using the *Alloy Output Specification* and the results are reported to the developer.

While this process involves quite a few entities, only the *TestEra Specification* needs to be created explicitly by the user. Everything else is automatically derived from it. Note that according to TestEra's authors, there are some cases where TestEra is unable to automatically derive the transformations. In these cases the user has to write them by hand. In our experiments we have not encountered one of these instances.

3.1.2 Workflow

TestEra provides a plug-in for the Eclipse IDE [2]. This makes it easy to use TestEra from a graphical user interface. This process is explained in section 3.1.1 and is divided into three steps which need to be performed by the user:

- **Step 1: Generating the Alloy Model**
  After writing the specification for the structure to be generated, the user has to press a button, which causes TestEra to create two Alloy files for the input and output specification.
- **Step 2: Generating JUnit Tests**
  Next, the test cases have to be generated. For this, the user presses another button which triggers the generation process. The Alloy Analyzer creates all valid instances within the specified bounds. For each instance a JUnit test case is created. Within the test case, the Java instance is created, the test method is executed and the instance is passed to the TestEra library in order to verify the postconditions. After this step, the user ends up with one JUnit test case for every instance of the specified structure.
- **Step 3: Running the Tests**
  Lastly, the user has to run the generated test cases. This can be done manually or the test cases can be added to a test suite and be run as part of a regression testing test suite.

```
1  class Node {
2
3     public Node left;
4     public Node right;
5  }
```

Figure 3.2: Node.java: Code for `Node` class.

```
1  @TestEra(invariant = {"all n: Node | n !in n.^(left + right)",
2                        "all n: Node | no n.left & n.right",
3                        "all n: Node | lone n.~(left + right)"})
4  public class BinaryTree {
5
6     public Node root;
7
8     @TestEra(preCondition  = {"all n: Node
9                               | n in this.root.*(left + right)" },
10             postCondition = {"true"},
11             runCommand    =  "1 BinaryTree, 4 Node")
12    public boolean testMethod() {
13       // ...
14    }
15 }
```

Figure 3.3: BinaryTree.java: TestEra Specification for `BinaryTree` class.

3.1.3 Usage Example

This section illustrates how TestEra can be used for test case input generation. In the interest of keeping the example simple, we will work with binary trees again. The trees are neither ordered nor do their nodes have any payload. Both features and other constraints on the tree could be easily added, but would unnecessarily complicate the example.

Figure 3.3 shows the Java source code for the `BinaryTree` class and fig. 3.2 the code for its `Node` class[2]. The `Node` class has two children: `left` (line 3) and `right` (line 4). The `BinaryTree` class has a reference (line 6) for the `root` of the tree. In addition, there is the `testMethod` method (line 12) in the `BinaryTree` class, which serves as an entry point for our tests.

To specify the structure to be generated, there are two annotations: For the `BinaryTree` in line 1 and for the `testMethod` in line 8. The first one describes the general structure for valid instances. The second one specifies a precondition and a postcondition, which is validated by TestEra after the method has been executed. Using the precondition, one can add additional constraints on the input for a single method. In addition, the `runCommand` keyword in line 11 adds bounds for the input.

The structure of the `BinaryTree` is described with three conditions within the annota-

[2]Both classes need to be in separate files, as TestEra does not support inner classes.

tion for the class:

- The first condition (line 1) tells TestEra that no `Node` is allowed to be its own (transitive) child. More programmatically: Descending the tree starting at any node may never lead to the same node again. Hence, this constraint prevents the tree from having cycles. The "^" operator denotes the transitive closure.
- The next condition (line 2) ensures (together with line 1) the uniqueness of nodes inside the tree. It tells TestEra that the `left` and `right` child of a node may not be the same.
- Lastly, the condition in line 3 constricts each node to have at most one parent. The `lone` keyword means: "zero or one" and the "~" operator denotes the transpose (or inverse in case of a binary relation).

In addition, there is a precondition for the `testMethod` method in line 8 and 9. It ensures that all used nodes are reachable from the `root` field in the `BinaryTree` class. The "*" operator denotes the reflexive-transitive closure. More details on the semantic of Alloy can be found in its specification [23].

Using the two files in fig. 3.2 and fig. 3.3 the Alloy Analyzer can generate all valid instances of the `BinaryTree`.

3.2 Transformations

As described in section 3.1, TestEra includes algorithms to transform instances between a Java representation and an Alloy one. The so-called concretization transforms an Alloy instance into a Java object. The abstraction can then be used to reverse this transformation. In addition, the specification written inside the Java annotations by the user has to be transformed into an Alloy model. TestEra also supplies an algorithm which can automatically achieve hat.

3.2.1 Specification Transformation

The first thing TestEra needs to do is to combine and translate the specification written by the user inside the Java annotations. Therefore, each annotated Java class gets translated into an Alloy signature. For every field a ternary relation is created within the Alloy model. Each tuple (o, n, s) of this relation assigns a value n to the field in an object o in a particular state s. TestEra uses two distinct states to model mutation on the object graph. One state is used to represent the state before the test method is executed and one for afterwards. This way the developer can access the pre-state in the postcondition and validate the changes made to the structure.

As an example consider the `BinaryTree` class in fig. 3.3. Its translation into Alloy code is shown here:

```
1 sig BinaryTree {
2    root : Node lone-> State
3 }
```

For this class, one signature is created. The `root` field, which is a reference to a `Node` object, is modeled as a ternary relation called `root`. As an example, assume it contains the following entries: $(b_1, n_1, PRE), (b_1, n_2, POST)$. This means that the `root` field was referencing the first node n_1 and is now (after the test method has been executed) referencing the second node n_2. The b_1 entry refers to the object which owns the field. In this case there is only one `BinaryTree` object as it is the root of the structure.

The constraints on the structure, specified by the user within Java annotations, are represented as a `fact` in Alloy. Alloy facts express constraints on existing types. They are simply copied by TestEra with some syntactical modifications.

In the `BinaryTree` example the three conditions are gathered in one `fact`:

```
1 fact BinaryTree_fact {
2    all s:State {
3       all n: Node | n !in n.^((left.s) + (right.s))
4       all n: Node | no n.(left.s) & n.(right.s)
5       all n: Node | lone n.~((left.s) + (right.s))
6    }
7 }
```

TestEra also models the method's parameters and its return value in order to allow the user to access them in the pre- and postconditions. To do this, TestEra simply introduces another ternary relation for each parameter and the return value. Obviously no return value exists in the pre-state (before method execution).

Continuing with the binary tree example, the parameters and return value is modeled like this:

```
1 static sig Pre extends State {
2    This: BinaryTree,
3    exampleParameter: Int
4 }
5
6 static sig Post extends State {
7     Result: Boolean
8 }
```

Line 1-4 define the parameters. For demonstration purposes we added another parameter, next to the implicit `this` reference. The second signature (`sig`) in line 6-8 models the return value.

3.2.2 Concretization

Once the Alloy Analyzer has created instances conforming to the given specification, TestEra needs to translate these instances into Java objects for the test case. Algorithm 6

shows how this transformation is achieved in pseudo code[3]:

```
   Input: alloyModel
   Output: The root of the java object graph.
1  map ⟵ ∅
2  foreach signature in alloyModel.signatures do
3  |  foreach atom in signature.atoms do
4  |  |  map ⟵ (atom.name, new signature.type)
5  |  end
6  end
7  foreach relation in alloyModel.relations do
8  |  foreach (from,to) in relation do
9  |  |  map [from ][relation.field] ← map[to ]
10 |  end
11 end
12 return map ["input"]
```

Algorithm 5: Alloy to Java

The algorithm works in two stages: First, it iterates over every atom[4] of all signatures. For each atom, a new map entry is created. The atom serves as a key and maps to a newly created Java object of the corresponding type.

After creating all objects needed for the test case, the second stage sets the references between these objects. Therefore, the algorithm loops over all relations in the alloyModel. Note that a relation defines all values for a specific field: A tuple "$(\mathsf{from}, \mathsf{to})$" of the relation expresses that the field of the relation in the "from" object references the "to" object. This relationship is translated into Java in line 9: First the "from" object is obtained from the previously created map. Next, the field specified by the relation is assigned to the "to" object, which is also retrieved from the map.

3.2.3 Abstraction

The last transformation TestEra needs is the abstraction. It is the inverse operation to the concretization: the algorithm transforms a Java object back into Alloy. This way TestEra is able to verify the postcondition using the Alloy Analyzer.

The algorithm basically has to walk the entire reachable structure and add every object as a signature to the Alloy model and every reference to the corresponding relation withing the Alloy model.

[3] Both, the concretization as well as the abstraction are implemented in Java in TestEra.
[4] Atoms in Alloy correspond to Java objects and Signatures to classes.

```
   Input: map, alloyModel, output
   Output: The new alloyModel representing the modified object graph.
1  visited ⟵ ∅
2  workSet ⟵ allMethodInputs()
3  if output == null then
4  |  alloyModel.setSig("Output", null)
5  else
6  |  alloyModel.setSig("Output", getAtom(map, output))
7  |  workSet ⟵ {getAtom(output)}
8  end
9  while workSet not empty do
10 |  obj ⟵ workSet.pop
11 |  visited ⟵ field.value
12 |  foreach field in obj do
13 |  |  target =getAtom(map, field.value)
14 |  |  alloyModel.getRelation(field.name).add(obj, target)
15 |  |  if field.value is a reference and not contained in visited then
16 |  |  |  workSet ⟵ getAtom(field.value)
17 |  end
18 end
19 return alloyModel
```

Algorithm 6: Java to Alloy

Therefore, the algorithm first constructs a workSet in line 1-8. The set contains all objects which could have any effect on the rest of the system, except static variables[5]: The input arguments including the implicit `this` reference (line 2) and the return value of the method (line 3-8).

Next, the algorithm walks through all parts of the object graph, which are reachable starting from the elements in the working set (line 9-18). This is done by simply extracting an element from the working set and looping over all of its fields. Each field value is recorded in the Alloy model using the previously explained relations. If the field is of a reference type, the referenced object is added to the working set, if it was not already visited.

The `getAtom` method tries to retrieves an Alloy atom from the map, which was created in the concretization. If it is not found in the map, it was created by the user's test method. In this case, a new Alloy atom is created and automatically added to the Alloy model as a new signature.

[5] Alloy ignores these, as they are discouraged in object oriented design.

3.3 Limitations

While working with TestEra, we encountered a few shortcomings. This is, as with Korat, to be expected when working with a research prototype. In the case of TestEra, the issues were not investigated a lot and this section only seves to document them.

3.3.1 Missing Language Support

There are a few Java language features which are not supported in TestEra. Without digging into the TestEra code base we can not give a reason why these features are not supported or if they could be supported at all.

- **Inheritance**
  Using inheritance to abstract common behavior and to decouple parts of the system is one of the core concepts of object oriented software design. The TestEra prototype does not support it, but tolerates it. An example shows what that means: Consider an inheritance structure, where `Derived` extends `Base`. It is possible to create instances of `Derived` using TestEra, as long as the specification does not refer to fields in `Base`[6].

  With inheritance being a given in a real world project, we propose a way of working around this limitation. If there are no fields in the super class which values need to be generated, TestEra can be used directly. Otherwise, the developer can extend the class to be tested and define each field of all super classes in this newly created class. The fields can then be generated by TestEra. In addition, the developer creates a wrapper method in the new class which first copies the values from the fake fields into the real fields of the super classes, then calls the method which actually should be tested and lastly copies the values back into the fake fields.

  This way, one could use TestEra for testing structures with inheritance. However, it involves additional manual labor and requires the fields of the super classes to not be private. In addition, this approach pollutes the project with a somewhat unnecessary class. But, on the plus side, this approach does not require the developer to write the TestEra specification inside a production class. Thus, keeping it clean of testing code.

- **Primitive Types**
  TestEra only supports positive integers and boolean values. The reason is that operations, such as mathematical and logical ones, have to be modeled within Alloy. Hence, this is not a conceptional problem but only a feature not yet implemented.

  A small experiment was conducted to get a rough idea of how important other primitive types are. Using a simple Linux bash script such as:

[6] According to the original authors of TestEra, the lack of inheritance support is not a conceptional issue, but was just not implemented for the prototype, see [28, p. 100].

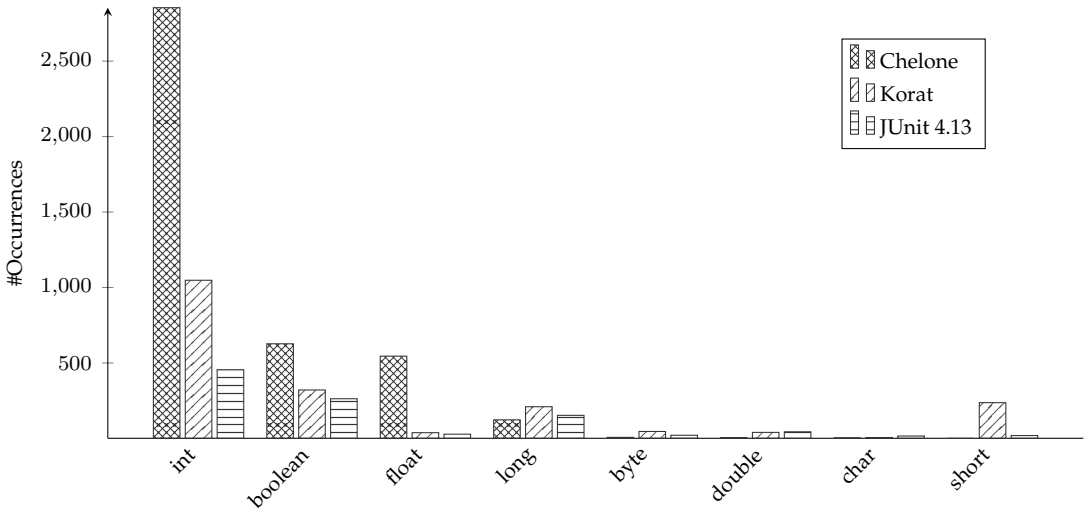


Figure 3.4: Primitive variable occurrences.

```
for f in {boolean,char,byte,short,int,long,float,double}; do
echo $f: `find . -name "*.java" -exec cat {} \;
          | grep "\W$f\W"
          | wc -l`;
done
```

one can easily obtain a fairly good estimate. The script simply iterates over all primitive types of Java and counts the number of lines in which they are contained. Note that this is not an exact measurement, but a fairly good estimate should be sufficient for this purpose.

The result is shown in fig. 3.4. The y axis shows the number of occurrences for the variable types in Java. Three different tools are shown: Chelone, Korat and JUnit[37]. For the first two we simply used the current build, as we are actively working on them. For the latter one, version 4.13 was used.

It is easy to see that integers and booleans are the most commonly used variable type. In Chelone there are also quite some floating point variables used. This is mostly due to the graphics code. In addition to graphics code, we expect floating pints primarily being used in scientific or statistical simulation code. But, as there is no good workaround[7] to this missing feature, we expect this to be very limiting in many places.

- **Nested Classes**
  In our experiments, TestEra was not able to deal with inner or static nested classes.

[7] In some places, the variable type could be changed into an integer.

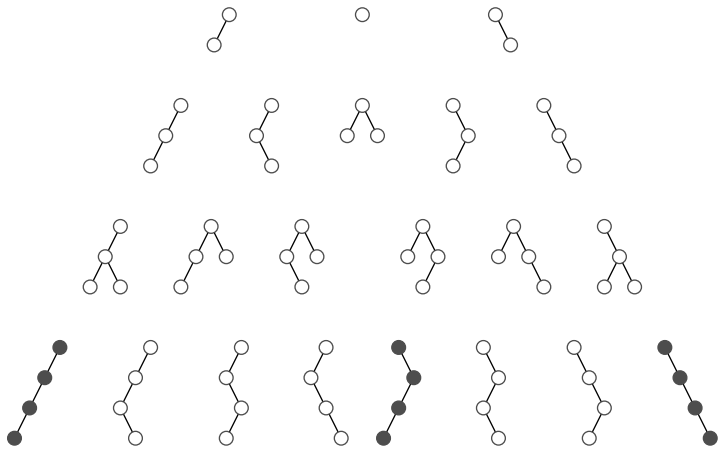


Figure 3.5: All binary trees with up to 4 nodes.

This issue was not mentioned by the original authors. Therefore, we do not know whether it is possible to implement this feature in TestEra or not, but it seems probable.

A possible workaround for static nested classes, which was used in the `BinaryTree` example in fig. 3.3, is to simply move the nested class to an upper level. In the case of inner classes, this refactoring becomes more difficult, as one needs to manually manage the reference to the outer class.

Refactoring only works for code which is able to change. Library code or code which has been shipped to customers can often not be changed. In these cases, the developer could – similar to the solution for the inheritance problem – create a wrapper class, which then initializes the actual class, including the inner classes.

These workarounds require quite a lot of refactoring or new code to be created, making it less attractive to use TestEra, if there are many nested classes in the project.

3.3.2 Isomorphism Problems

During our experiments, we encountered an issue where isomorphic instances were being generated. Note hat this is not a problem with TestEra itself, as it is the responsibility of the Alloy Analyzer to generate test cases and therefore also to filter isomorphic ones.

The issue occurs in the binary tree example we introduced in fig. 3.3. When generating trees with 4 nodes, TestEra generates 26 trees[8]. The formula for the number of binary trees with up to n nodes $f(n)$, which was derived in section 2.2.1.1, states that there are only 23 trees. Hence, TestEra generates three isomorphic trees.

[8]This issue also occurs for larger trees.

Figure 3.5 shows all binary trees with up to 4 nodes. There are 22, as we did not visualize the empty tree with zero nodes. TestEra allows the user to visualize the instances it generates using Alloy's visualization capabilities. Using this feature and fig. 3.5, one can easily find the isomorphic ones: They are drawn with filled black nodes.

3.3.3 Encapsulation Violations

Inside the generated test cases, the required objects are simple created and their references are directly assigned:

```
1 BinaryTree BinaryTree_0 = new BinaryTree();
2 Node Node_0 = new Node();
3 BinaryTree_0.root = Node_0;
```

This is a very fast process to create a test structure, but it also implies a problem: The `root` field of the `BinaryTree` needs to be accessible from withing the test function. Therefore, it needs to have `public` visibility or, assuming the test function is in the same package as the `BinaryTree`, have `protected` or `default` visibility. In any case, this would usually force the developer to escalate the visibility of the classe's fields.

While this violates a core principle of object oriented software design, the problem is not specific to TestEra: When performing the unit testing manually, one would also need to create the `BinaryTree` object. There are two ways of doing this: Either access the fields directly, which leads to exactly the same problem, or constructing the object via method calls. This leads to a unit test, which depends on several methods and no longer tests one method in isolation. In addition, the developer no longer has precise control over the details of the created structure.

Note that this problem can not be fixed by using reflection to assign values to `private` fields. Doing so would allow the developer to keep the `private` visibility officially. But, in actuality it creates an even worse problem: There is an invisible dependency on the field now. Hence, no developer looking at the class at hand would ever assume that this field is accessed by anyone else but the class itself. It just obscures a dependency and therefore makes it easier to break.

In general, it would be impossible to make TestEra use a sequence of method calls to create an object graph. Therefore, the approach taken (assigning fields directly), seems like a valid tradeoff.

4 JCute

JCute [39] is a **C**oncolic **U**nit **T**esting **E**ngine for **J**ava programs. It was developed around 2006 at the Open Systems Laboratory of the University of Illinois. Before creating JCute, they developed a similar concolic testing tool for C programs: Cute [40]. It provides the same functionality as JCute, but does not allow to test multi threaded code. With this thesis being focused on testing Java programs, only JCute is considered here.

This chapter is laid out as follows: First, an overview of JCute from a user's perspective is given in section 4.1. After a short introduction, a simple example is used to explain how JCute works. To make it better comparable to Korat and TestEra, the section also includes a second example with a binary tree. Lastly, in section 4.2 the inner workings of JCute are explained.

4.1 Description

As described in chapter 1, JCute is a concolic testing tool for Java. It builds on the work of [29], who first introduced concolic testing. The general idea is to start off with random input. During the concrete execution of the method under test, symbolic constraints are gathered and used to derive new test inputs. The goal is to provide a high coverage with very few test cases by only generating ones with different control flows through the code under test.

This section first introduces JCute from a high level perspective in section 4.1.1. Next, a simple example method is used to illustrate how one can use JCute and how JCute derives the test cases in section 4.1.2. In order to make JCute better comparable to Korat (chapter 2) and TestEra (chapter 3), section 4.1.3 shows how to generate binary trees with JCute.

4.1.1 Overview

In order for JCute to track the input symbolically, the code under test first has to be instrumented. JCute uses the SOOT compiler framework [44] to achieve that: First, the code is compiled into byte code using a standard Java compiler. Then, the SOOT compiler framework is used to instrument the code. For each instruction, calls to the JCute runtime environment are added. These calls provide the necessary hooks for JCute to record the imposed constraints on the input variables during runtime.

Once the code has been compiled and instrumented, it is ready to be tested by JCute. Figure 4.1 shows the main loop of JCute: First, random values are generated for all input

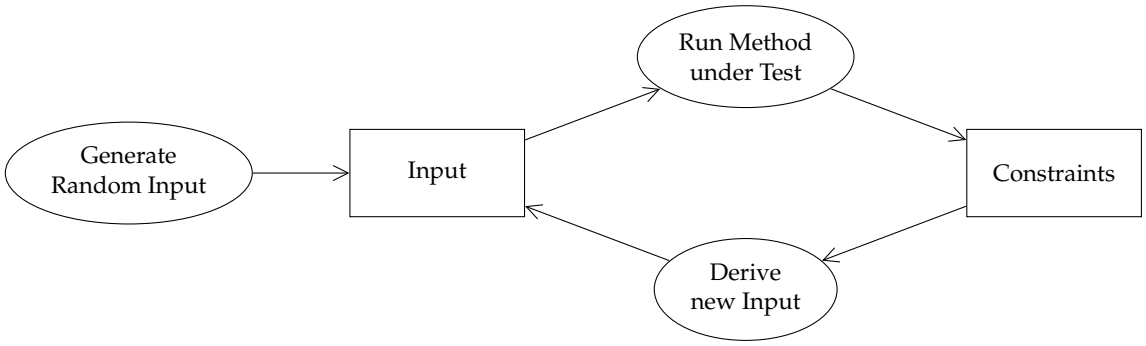


Figure 4.1: Overview of JCute.

variables. JCute has not gained any knowledge on the code to be tested yet, therefore random is the best it can do in the beginning. Next, the method under test is executed with these values and the symbolic constraints are tracked during the execution. The gathered constraints describe one specific path through the method. By negating some of the predicates, a formula describing the properties for an input for a different path can be obtained.

This formula is solved using a custom constraint solver, which provides several problem-specific optimizations (see section 4.2.3). It also has some limitations: For example only linear equation systems can be solved. Therefore, whenever a non-linear symbolic expression is encountered, it is replaced with the concrete value from the current execution. This makes the found solutions less precise, but allows for a more efficient constraint solver.

Using the newly obtained input, the method is run again. This process is repeated as long as JCute is able to find new modifications on the formula.

Each method can be visualized as a control flow graph. Each node in the graph represents a decision (conditional jump), which is based on the input. Note that a condition could also be constant (like "$a == a$") or the decision could be based on static variables. The latter case is ignored by JCute. In all other cases the decision is based on the method's input or the implicit `this` argument. Therefore, every decision represented by the nodes of control flow graph is dependent on the input and the outcome can be changed by alternating the input[1]. JCute basically uses backtracking on these nodes by negating some of the constraints to explore all possible paths of the control flow graph[2].

4.1.2 Simple Example

Listing 4.1 shows an example Java program, which is used to illustrate how JCute works. For simplicity, all functionality is gathered within one class in one file. In a real world

[1] There may also be decision outcomes which can never be reached.

[2] As there can be a very large number of paths for methods with loops or jumps, a maximum search depth is used.

```
1  public class SimpleExample {
2
3     static void testMe(int a, int b) {
4        if(a != 0) {
5           if(b != 0) {
6              int x = 3 * b;
7              if(x == a) {
8                 Cute.Assert(false); // -> error
9              }
10          }
11       }
12    }
13
14    public static void main(String[] args) {
15       int a = Cute.input.Integer();
16       int b = Cute.input.Integer();
17
18       testMe(a, b);
19    }
20 }
```

Listing 4.1: Simple test example for JCute.

setup, the test code can be contained in a different file, thus separating production and test code.

The `main` method in line 14 serves as an entry point. First, the JCute API is used to create two integer values (line 15 and 16). These two variables serve as an input for the `testMe` method and they are tracked symbolically by JCute. As previously stated, JCute will use random values in the first iteration. Unlike Cute, JCute initializes the primitive values with `0` instead of using a random number generator. Thus making the process less random, but still independent from the code under test. JCute is also able to generate objects. These reference values are always initialized with `null`, just like with pointer values in Cute.

In line 18 the test method (`testMe`) is called using the two generated variables `a` and `b`. This function is mainly written for demonstration purposes. First, it checks that both, `a` and `b`, are not equal to zero. Then it checks if `a` is equal to three times `b`, if so the program is terminated by an assertion. Otherwise, the method simply returns.

Note that there are four distinct paths through the method. In this case all paths are possible and therefore, JCute is able to explore all of them. Figure 4.2 shows the chosen values for each iteration:

- **1. Iteration**
  As already stated, JCute chooses zero for all primitive values in the first iteration. Calling the `testMe` method with `a=0` and `b=0` will cause the execution to take the `else`-branch of the first `if` statement (line 4) and immediately return. Let a_0 and b_0 denote the symbolic variables representing the respective concrete variables. JCute gathers the constraints for the path taken through the control flow graph in a for-

	1. Iteration	2. Iteration	3. Iteration	4. Iteration
a	0	1	1	3
b	0	0	1	1
path constraint	$a_0 = 0$	$a_0 \neq 0 \wedge b_0 = 0$	$a_0 \neq 0 \wedge b_0 \neq 0 \wedge x_0 = 3 \cdot b_0 \wedge x_0 \neq a_0$	$a_0 \neq 0 \wedge b_0 \neq 0 \wedge x_0 = 3 \cdot b_0 \wedge x_0 = a_0$

Figure 4.2: Symbolically tacked variables.

mula, which is called *path constraint*. It defines the characteristics of all inputs leading to the same execution path[3]. In this case, only one constraint is gathered for the jump to the `else`-branch in line 4. Hence, the following path constraint: $\langle a = 0 \rangle$.

- **2. Iteration**
  In order to generate the input values for the second iteration, JCute modifies the path constraint by negating the last predicate. This leads to the new path constraint: $\langle a_0 \neq 0 \rangle$, which will cause the program to take the other branch for the `if` statement, which was executed last.

  Using a constraint solver JCute is able to find concrete values for the symbolic constraints. As shown in fig. 4.2, only the value for the variable `a` has changed, as there are no constraints for `b`. Executing the `testMe` method with these new values causes the execution to take the `then`-branch for the first `if` statement (line 4). The next `if` statement (line 5), which checks that `b` is positive, fails and the method returns, leading to the new path constraint: $\langle a_0 \neq 0 \wedge b_0 = 0 \rangle$.

- **3. Iteration**
  For the next iteration JCute negates the last predicate of the path constraint again, leading to: $\langle a_0 \neq 0 \wedge b_0 \neq 0 \rangle$. A possible solution for the equation is $a_0 = 1$ and $b_0 = 1$. Using these values for the concrete variables leads to an alternated execution path where the `then`-branch of the first two `if` statements are taken.

  In the following two lines (line 6-7), the `testMe` method checks if `a` is equal to three times `b` by using a local variable `x`. For the input $a_0 = 1$ and $b_0 = 1$, this is not the case and the method simply returns.

  The `if` statement in line 7 depends on `a` and `x`, which depends on `b`. JCute is able to track this transitive dependency of the `if` statement on `b` and to derive the following path constraint: $\langle a_0 \neq 0 \wedge b_0 \neq 0 \wedge x_0 = 3 \cdot b_0 \wedge x_0 \neq a_0 \rangle$.

- **4. Iteration**
  For the last iteration, JCute inverts the last predicate of the path constraint again, leading to: $a_0 \neq 0 \wedge b_0 \neq 0 \wedge x_0 = 3 \cdot b_0 \wedge x_0 = a_0$. This formula is solved by the

[3] Assuming a deterministic program.

constraint solver, which leads to the concrete values: `a=3` and `b=1`, as `a` needs to be equal to three times `b` to reach the assertion.

When executing the `testMe` method again, another distinct path through the control flow graph is taken, which takes the `then`-branch of the inner most `if` statement and triggers the assertion. JCute reports this error and provides the developer with a concrete set of input values, which trigger the program fault. This makes it easy for the developer to reproduce the problem and debug it. Note that this error, as all errors found by JCute, is a real error. It was found during the concrete execution and not by some symbolic evaluation.

After this iteration JCute is done with its exploration. Negating the last predicate again would lead to a path constraint, which was already explored in iteration 3. Therefore, JCute has now explored all four possible paths through the method leading to a path coverage of 100%.

4.1.3 Binary Tree Example

JCute can also be used to generate complex data-structures. Unlike with specification-based testing tools, like Korat (chapter 2) or TestEra (chapter 3), the structure is not specified explicitly. In order to generate a data-structure, successive calls to its methods are used. For example, to create a binary tree with four nodes, one would simply insert four nodes into an empty binary tree. As shown in section 4.1.2, JCute tries to explore all possible different paths through a method. Therefore, JCute explores all paths through the insert method of the data-structure and thereby creates different instances.

Unlike with the specification-based testing tools, this approach does not guarantee to find all possible instances. Instead, it tries to find instances which take different paths through the creation methods. Assuming the method under test is called right after the data-structure has been created, the execution still runs within the JCute runtime and therefore JCute also tries to find different paths through this method.

Listing 4.2 shows how one can test a binary tree using JCute. Unlike in specification-based testing, we need to use a more realistic tree, where the `Node` objects hold a payload. Hence, we are using a binary search tree. The reason for this is that JCute uses the API of the `BinaryTree` class to generate the trees. In order to write an insert method, we need to have some value associated with a node to be able to insert it at the right place. Otherwise, there would be no reasonable way to construct the tree.

The binary search tree maintains one simple invariant: All payloads in the left sub-tree of a node are smaller than its payload and vise versa for the right sub-tree. In addition, each payload is unique. Hence, there can not be two nodes holding the same number.

In the example in listing 4.2, line 3-8 declares the `Node` object with its associated `value` and a reference to the `left` and `right` child. In addition, there is a constructor to initialize the `Node` object. The `BinaryTree` class has a reference to the `root` of the tree and offers an `insert` method. We do not show the implementation for this method to keep the

```
1  public class BinaryTree {
2
3     public static class Node {
4        Node(int value) {this.value = value;}
5        int value;
6        Node left;
7        Node right;
8     }
9
10    private Node root;
11
12    void insert(Node node) { /* ... */ }
13
14    public static void main(String[] args) {
15       BinaryTree tree = new BinaryTree();
16       for(int i=0; i<3; i++) {
17          tree.insert(new Node(Cute.input.Integer()));
18       }
19
20       Test.testBinaryTree(tree).
21    }
22 }
```

Listing 4.2: Simple test example for JCute.

example clear. When inserting a node n, the method simply walks down the tree until it finds a valid position to insert n. At each node c there are three options: (1) If the two nodes' payloads are equal, the search is completed and n can be discarded, as it is already contained in the tree. (2 & 3) If the payload of c is smaller (larger) than the payload of n, the search continues at the left (right) child of c. If there is no left (right) child n, becomes the left (right) child of c.

Just like in the simple example in section 4.1.2, the execution starts at the main function (line 14). First, a `BinaryTree` object is created (line 15) and 3 nodes are added (line 16-18). After that, the user can call any method to perform various tests on the tree (line 20).

When running the code with JCute, all structurally different binary trees with 1, 2 and 3 nodes are created. For this example, JCute creates 13 trees. Due to the payload, there are 5 trees which are structurally equal to the other ones and therefore redundant for our purpose. This leaves 8 created trees, which are structurally different. This number can be verified using the formula in section 2.2.1.1: $f(3) = 9$. Note that the code in listing 4.2 does not generate the empty tree with zero nodes, therefore there are only 8 trees instead of 9 trees.

We also experimentally verified that JCute creates all structurally different binary trees for up to 4 nodes. In this case, JCute creates a total of 75 trees and only 22 are structurally different. As we have to filter out the redundant trees manually, we did not do the verification for 5 nodes, which yields 541 structures.

4.2 Algorithm

In this section we lay out the interesting parts of the algorithm and the techniques used in JCute. Next to the concrete execution, the program under test is executed symbolically. Therefore, the code is instrumented and each instruction is monitored by JCute. In order to build constraints on the input variables and derive new inputs, the program state is kept in a symbolic state as well. This state is updated each time the concrete state changes. At the end of the execution the constraint solver uses it to derive a new test input.

First, the entities of the symbolic state are described in section 4.2.1. After that, section 4.2.2 explains how the code is instrumented and describes how the changes to the concrete state are transfered into the symbolic state. Section 4.2.3 then outlines the main properties of the constraint solver used by JCute. Lastly, section 4.2.4 explains how JCute offers support for multi-threaded code in Java.

Note that this description is mainly based on Cute [40], as the paper on JCute [39] does not provide a detailed description of JCute. But with JCute being based on Cute, the concepts are very similar. All details about Cute can be found in [41].

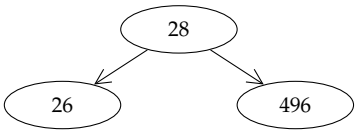
4.2.1 State Representation

JCute uses several data-structures to symbolically monitor the execution of the program at runtime. The following structures are used:

- **Logical Input Map $\mathcal{I}$**
  The logical input map $\mathcal{I}$ is a symbolic representation of the current object graph in the program. It stores all primitive values and references between objects. Instead of storing the physical addresses of the involved objects, logical addresses (represented by numbers in $\mathbb{N}$) are used. This indirection has the advantage that the logical address of an object does not change between two executions. The physical address, however, could change each time the object gets allocated. In addition, the usage of logical addresses allows JCute to make consecutive inputs similar [40].

  $\mathcal{I}$ is a partial function: $\mathcal{I} : \mathbb{N} \mapsto \mathbb{N} \cup \mathbb{V}$. Where $\mathbb{N}$ are logical addresses and $\mathbb{V}$ is the set of all values of all primitive data types. This structure is updated each time the object graph of the actual program is updated. It can be used to represent any data-structure.

  To give an idea of how such a map looks like, consider fig. 4.3 as an example. It shows a binary tree (as described in listing 4.2) and its logical input map $\mathcal{I}$. The first row of the table shows the logical address and the second row the associated value or logical address. The first column (logical address 0) represents the `root` field of the binary tree. It is a reference to a `Node` object at the logical address 1. Starting at logical address 1, the logical input map consecutively contains the fields of the root of the tree: `value`, `left` and `right`. The other two `Node` objects reside at logical address 4 and 7.



$\mathcal{I} =$

0	1	2	3	4	5	6	7	8	9
1	28	4	6	26	0	0	496	0	0

Figure 4.3: Representation of a binary tree as a logical input map $\mathcal{I}$.

- **Memory Map** $\mathcal{M}$
  This map is used to bridge the gap between symbolic variables and concrete ones. It maps each logical address to a physical variable. Therefore, JCute is able to find the concrete variable for any symbolic variable in $\mathcal{I}$.

- **Predicates** $\mathcal{P}$
  This structure maps variables to arithmetic constraints on primitive and reference values. These are gathered at each statement which changes the concrete state. They are linear constraints, i.e. of the form: $a_1x_1 + a_2x_2 + \cdots + a_nx_n + c \bowtie 0$, where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$. Non-linear expressions in the program are not tracked as symbolical constraints, instead their concrete value is used directly. Note that JCute does not track constraints with $n = 0$, as these are constants, which can also be obtained from the concrete state.

  For reference values the tracked constraints are simpler, as there is no pointer arithmetic in Java. They are of the form $x \cong y$ or $x \cong \texttt{null}$, where $\cong \in \{=, \neq\}$.

4.2.2 Instrumentation & Execution

The instrumentation inserts calls to the JCute runtime into the program. This allows JCute to track the variables symbolically. First, the Java code is compiled into byte code with a standard Java compiler. After that, the SOOT compiler framework [44] is used to first simplify statements by introducing temporary local variables and thereby splitting complex calculations into several smaller steps. This makes the second step, the instrumentation, easier, as only very basic statements have to be instrumented by adding calls to the JCute API. Three functions are needed for the instrumentation: `initInput`, `executeSymbolic` and `executePredicate`. Basically, `initIput` initializes a new variable, `executeSymbolic` does an assignment and `executePredicate` evaluates the condition for a branch. These three methods are explained in the following sections.

There is no description of any of the algorithm for JCute by the original authors. Therefore, the following explanation only shows how we think that the algorithm described for Cute would look like for JCute. Even though it might be implemented slightly differently, the general idea should be the same.

4.2.2.1 Method: `initInput`

The `initInput` method is used every time the user requests a value for a variable from JCute. In the previous examples (listing 4.1 and listing 4.2) this was done via a call to the `cute.input.Integer()` method. It is also possible to request variables of different data types or reference values. In all cases, JCute initializes the variable using the logical input map. The details are shown in alg. 7.

```
   Input: variable              // to be initialized
   Input: logicalAddress        // of the variable
1  if logicalAddress ∉ 𝓘 then
2      variable ⟵ 0
3      𝓘 ⟵ 𝓘[logicalAddress ↦ 0]
4  else
5      v ⟵ 𝓘(logicalAddress)
6      if typeOf(v) is primitive value or v == 0 then
7          variable ⟵ v
8      else
9          if v ∈ 𝓜 then
10             variable ⟵ 𝓜(v)
11         else
12             h' ⟵ h
13             variable ⟵ allocate(typeOf(v))
14             𝓘 ⟵ 𝓘[logicalAddress ↦ h']
15             𝓜 ⟵ 𝓜[h' ↦ variable]
16             foreach field in variable do
17                 initInput(field, h'++)
18             end
19         end
20     end
21 end
22 𝓟 ⟵ 𝓟[variable ↦ x_l]
```

Algorithm 7: `initInput` Method in pseudo code.

The algorithm gets the logical address and the variable as an input. For simplicity we do not consider Java reflection techniques for maintaining a reference to the variable and setting it. We simply assume that we have some kind of handle to the variable, which allows us to assign values to it and obtain its type information.

At the beginning (line 1), the algorithm checks whether the logicalAddress is defined in the logical input map $\mathcal{I}$. If it is not defined, JCute has never seen the variable before and

therefore initializes it to zero (line 2). This update is also kept in the logical input map $\mathcal{I}$ (line 3).

Otherwise (line 5), if the logicalAddress is defined in $\mathcal{I}$, JCute has already decided on a value for the variable. This value is read from the logical input map $\mathcal{I}$ in line 5. If it is a primitive type or if it is a null reference, JCute does not need to allocate any objects and can directly assign the value to the variable (line 7).

Otherwise (line 9), the algorithm now knows that the variable should be initialized and pointed to an object (not `null`). The next question (line 9) is whether this referenced object was already created in the current execution. This can happen, if two variables point to the same object and the other one has already been initialized. If the object was already created (line 10), it can be obtained using $\mathcal{M}$, which maps logical addresses to concrete objects.

Otherwise (line 12), if the object does not exist already, a new object needs to be allocated and initialized. The global variable h holds the next free logical address, which is the place where the algorithm puts the object to be created in the logical address space. The `allocate` function creates an object and increases h by the object's size (number of its fields). The algorithm keeps the initial value of h in h' in order to be able to address the fields of the object later. Next, the new object is added to $\mathcal{I}$ and $\mathcal{M}$. After that, each field of the newly created object is initialized by recursively invoking the `initInput` method.

Using this method, JCute is able to create concrete values and objects using the logical representation $\mathcal{I}$ of the object graph, which is created by the constraint solver.

4.2.2.2 Method: `executeSymbolic`

A call to this method is inserted by the instrumentation framework whenever a value is assigned to a variable. Its purpose is to transfer these changes from the concrete state into the symbolic state. It is used for primitive variables as well as for reference variables. Algorithm 8 shows the pseudo code for the algorithm.

As an input, the algorithm gets a x and an expression e. In the concrete code, the expression e is assigned to the x. The algorithm needs to do the same for the symbolic state. The algorithm uses a **switch**-statement to treat the different types of expressions. Note that there are only unary or binary expressions, due to the simplifications made by the SOOT compiler:

- **Unary Expressions**
  In this case, another variable v_2 is assigned to the x. Hence, JCute needs to add a predicate to the set of predicates $\mathcal{P}$, which expresses that x has the same value as v_2. Therefore, JCute first does a lookup to check if v_2 is contained in $\mathcal{P}$. If this is the case, a predicate is added to $\mathcal{P}$ (line 3), which reflects the equality.

  Otherwise, there are no constraints on v_2 in the current symbolic state. Therefore, the value for v_2 can only be obtained from the concrete state. JCute could add a constraint that expresses that x is equal to this concrete value. But as previously

```
   Input: x        // which is getting assigned
   Input: e     // which is assigned to x
1  switch e do
2  |  case "v1 ":
3  |  |  if v1 ∈ P then              P ⟵ P[x ↦ P(v1)]
4  |  |  else                       P ⟵ P − x
5  |  end
6  |  case "v1 ± v2":
7  |  |  if v1 ∈ P and v2 ∈ P then  P ⟵ P[x ↦ P(v1) ± P(v2)]
8  |  |  else if v1 ∈ P then        P ⟵ P[x ↦ P(v1) ± v2]
9  |  |  else if v2 ∈ P then        P ⟵ P[x ↦ v1 ± P(v2)]
10 |  |  else                       P ⟵ P − x
11 |  end
12 |  case "v1 · v2":
13 |  |  if v1 ∈ P then              P ⟵ P[x ↦ P(v1) · v2]
14 |  |  else if v2 ∈ P then        P ⟵ P[x ↦ v1 · P(v2)]
15 |  |  else                       P ⟵ P − x
16 |  end
17 |  otherwise
18 |  |                             P ⟵ P − x
19 |  end
20 endsw
```

Algorithm 8: `executeSymbolic` Method in pseudo code.

mentioned, JCute does not keep constant expressions in $\mathcal{P}$, as these can always be obtained from the concrete state. To reflect this in the symbolic state, the variable x is removed from $\mathcal{P}$ in order to delete predicates which were gathered earlier.

- **Linear Binary Expressions**
  These expressions are treated in a similar fashion: If both operands are contained in the symbolic state, a new predicate is added, which uses these symbolic variables. If one of the two operands is not contained in the symbolic state, its concrete value is used. Otherwise, if neither of the operands has any predicates in $\mathcal{P}$, the variable x is removed from the symbolic state.

- **Multiplication Expressions**
  The constraint solver used in JCute is not able to handle non-linear equation systems. Therefore, JCute can not add a predicate containing a multiplication of two symbolic variables. Hence, JCute replaces at least one of the symbolic variables by its concrete value. If neither of the operands is contained in the symbolic state, the variable x is

removed again.

- **Other Non-linear Binary Expressions**
  For other expressions, like division or modulo, JCute can not add any constraints to the symbolic state. Therefore, the variable x is removed from the symbolic state in order to clear any previously added constraints.

4.2.2.3 Method: `executePredicate`

The last method needed by JCute does not propagate changes from the concrete state into the symbolic state. Instead, it is used to keep track of the outcome of all branches in the program under test. A branch is an if statement[4], which depends on a condition. This condition is transformed into a symbolic predicate and stored in the path constraints. Algorithm 9 shows the pseudo code.

```
   Input: v1 ⋈ v2          // which is getting assigned
   Input: concreteResult   // which is assigned to
1  condition ⟵ true
2  if v1 ∈ P and v2 ∈ P then  condition ⟵ P(v1) ⋈ P(v2)
3  else if v1 ∈ P then        condition ⟵ P(v1) ⋈ v2
4  else if v2 ∈ P then        condition ⟵ v1 ⋈ P(v2)
5  else                       condition ⟵ concreteResult
6  if neg(concreteResult) then condition ⟵ neg(condition)
7  pathConstraint [i] ⟵ condition
```

Algorithm 9: `executePredicate` Method in pseudo code.

The algorithm receives the predicate expression and the result of the predicate in the concrete execution as an input. Similar to the `executeSymbolic`, the expression is extracted into the symbolic state depending on the availability of its operands: If both operands are contained in the symbolic state $\mathcal{P}$, they are used directly. If one of them is available, its concrete value is used instead. Otherwise, if none of the operands is available, JCute simple uses the result value of the predicate in the concrete execution. The resulting symbolical constraint is stored in the condition variable.

Next, in line 6 and 7 the algorithm stores the extracted symbolic predicate (contained in the condition variable) in the path constraint. As the path constraint describes the path which was taken in the concrete execution, JCute needs to negate the constraint in case the condition evaluated to false. As an example, consider an if statement which checks

[4] If statements are the only branches, because switch statements are simplified into a chain of if statements by the SOOT compiler framework.

for "`x == 0`". If `x` is not equal to `0`, the else-branch is taken and therefore the constraint describing the path which was taken is the negation of the condition: "`x!=0`".

4.2.3 Constraint Solver Tweaks

After each run of the program, the path constraint contains all predicates gathered at each branching point during the execution. The last predicate in the path constraint is negated[5] to create a new path constraint which leads (likely) to an alternative path through the method.

The resulting formula needs to be solved. To achieve this in an efficient manner, JCute uses the lp_solve [7] tool as a basis. It is a library for solving linear integer equation systems. Just like all other libraries for a broad spectrum of problems, it is reasonably fast for many different cases, but it can be extremely optimized for one specific kind of problem. JCute introduces three tweaks to improve the efficiency for the problem at hand:

- **Fast Unsatisfiability Check**
  Before lp_solve is invoked, JCute first does a quick check, if the equation system is satisfiable. This is performed on a syntactical level: JCute checks, if the last predicate in the path constraint, which was negated, is the negation of any of the other predicates. If this is the case, the equation system is not satisfiable. According to the authors, this filter reduces the number of following checks and semantic evaluations by 60-95%.

- **Common Sub-Constraints Elimination**
  JCute analyses the formula and tries to remove common sub-constraints. Doing so reduces the size of the equation system, which allows for more efficient solving. This optimization and the next one together reduce the number of sub-constraints in the equation system by 64-90%.

- **Incremental Solving**
  Next to increasing the performance, this last optimization also helps to keep the subsequently found solutions more similar, which is helpful for the developer looking at the generated test cases. The idea is to identify the parts of the formula which have changed during two executions and reuse the parts which are independent of the changes. For details on this optimization see [40].

4.2.4 Concurrency

JCute allows for testing concurrent programs. Similar to the concept for single threaded programs, where JCute tries to explore all different paths through the control flow graph, in multi threaded programs, JCute tries to find all different scheduling orders, thus exploring possible data-races and deadlocks.

[5] More predicates are negated using backtracking, if the resulting path constraint was already explored.

To achieve this, a second structure, similar to the path constraint structure, which keeps track of all decisions in the program, is introduced. This schedule orders all events, like shared data reads and writes, between threads in the program. At the end of each execution, either the schedule or the path constraint is changed. This leads to a different scheduling between the threads or an alternative execution path through the program.

The scheduling between the threads is controlled by further instrumenting the code during compilation: Before each statement which accesses shared data, a function call to the JCute runtime is added. This gives JCute the chance to stop the thread. JCute can now pause the execution until the thread is allowed to run again according to the previously created schedule.

This feature has proven itself extremely useful when testing concurrent data structures. It has the advantage that it explores all possible schedules, which is unlikely to happen in manually written tests. By making the schedule deterministic, it is likely to find bugs which occur only very rarely in a real execution, where the scheduling between threads is done by the operating system. [39] shows, how JCute was able to find a large number of bugs in several of the collection classes of the JDK 1.4. For instance, testing the HashSet implementation revealed 19 data races and 9 deadlocks.

5 Experiments and Evaluation

This chapter contains the most interesting of the experiments for automated test input generation which were conducted throughout this thesis. It showcases how the introduced tools can be used to generate test inputs and thereby demonstrates their strengths and weaknesses. The experiments where all conducted in the context of the hobby game project *Chelone*. Therefore, this chapter gives an overview of how automated test input generation can be used in a real world project. Most of the described test cases with Korat are now part of *Chelone's* regression test suite and have already proven themselves useful.

With Korat being the main focus of this thesis, most experiments are conducted using Korat. Finding places to use the other tools has proven itself to be a difficult task: TestEra's incompatibility with inheritance is a major issue here. Rewriting the original code is often not an option in a real world project, as it would destroy or worsen the project's architecture. The other workarounds described in section 3.3 generate a lot of duplicated, highly coupled code, which is also not desirable. While JCute is more compatible with the *Chelone* code base, it is often more difficult to write the specification, as shown in section 5.4.

The following two sections describe the test environment (section 5.1) and the hobby game project *Chelone* (section 5.2) briefly. After that, each section describes one of the experiments.

5.1 Environment

All experiments where conducted on a MacBook Pro (late 2013)[1]. It has a 2.8 GHz dual-core Intel Core i7 processor and offers 16 GB DDR3 RAM. As an operating system, the machine runs OS X 10.10.4 (Yosemite).

For the experiments, which compare the performance of the testing frameworks, a virtual machine is used, as it makes the installation of TestEra much simpler. In addition, we are only interested in comparing the testing frameworks. Therefore, only their relative performances matter. The virtual machine is hosted inside an Oracle VirtualBox. It runs a Linux Mint 17.1 and has version 1.7 of the OpenJDK installed.

As we couldn't afford an expensive high-end server, we used a commodity class desktop computer to show that the performance inside the virtual machine is comparable to that of an average server. The desktop computer runs Windows 10 on a Intel Core i5 2005k @3.3GHz with 16 GB of DDR3 RAM. The machine has an Oracle JDK version 1.7 installed.

[1] Details about the machine can be found here: `https://support.apple.com/kb/SP691`.

5.2 Sample Project: Chelone

The following experiments were conducted within the context of a hobby game project: *Chelone* is a classic 2d orthogonal role playing game for mobile phones. As common in these games, the player has to control a group of heroes and save the world from an evil power. The player can walk through various regions of the game world, talk to characters in the game and fight monsters.

Chelone tries to set itself apart from other games by having a large, interactive and consistent game world. This is achieved by a powerful editor, which can be used to create all components of the game world: regions, dialogs, monsters, quest, items and so on. This approach allows the actual game engine to be mostly independent of the actual game. Therefore, it can be used for creating follow-ups or other similar games.

In order to create an interactive world, the editor includes a custom scripting language which lets the level designer control most aspects of the game: move characters around, play animations or start dialogs. In addition, it includes commands for conditional jumps and function calls. With this powerful scripting language, the game world itself can be seen as a program which is interpreted by the game engine.

The complexity of the game world results in a very large number of possible inputs for the game engine. In this thesis, we try using the introduced tools for automatic test input generation to create various parts of the game world and use them to test the interpreter: i.e. the game engine.

Both the game itself and the editor are written in Java. Together, they make up roughly 60.000 lines of source code. The database, which is used by the editor to store the game world, contains about 150 tables. The game world currently requires approximately 25.000 database entries and offers a player around 1 to 2 hours of play time.

For each of the individual experiments, we give a more detailed explanation of the respective part of the game.

5.3 Experiment: Expression Tree Serialization

As already mentioned, *Chelone* incorporates a powerful scripting language. It is needed to program all kinds of events in the game world. The script also includes mathematical expressions, which can be used to assign values to variables or for predicates in conditions. As an example, assume the game character gets a quest to collect a number of flowers in order to brew some kind of potion. This could be implemented by using a variable to count how many flowers the player has already collected. To do so, the developer can set up a trigger which starts the execution of a script each time a flower is picked up. The script then increments the flower variable and checks whether its value is large enough. Once a certain number of flowers has been collected, a dialog could be started to tell the player that he has collected enough flowers.

To implement these mathematical expressions, a so-called expression tree is used. This

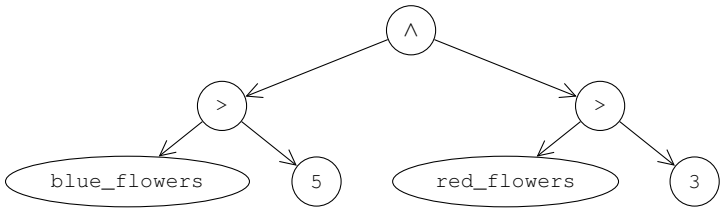


Figure 5.1: Expression tree for a condition.

tree can currently contain binary expression (like arithmetic operations or comparisons), unary expressions (like negations) or nullary expressions (like variable accesses, constants or random values). It could also be extended in the future to have other special purpose expressions. As an example, consider the tree shown in fig. 5.1. It the syntax tree for a condition which evaluates to `true` if and only if the `blue_flowers` variable is larger than `5` and the `red_flowers` variable is larger than `3`.

The evaluation of the expression trees is implemented using the interpreter pattern [18]: There is an abstract expression class from which every concrete expression class inherits. The children of a concrete expression are references to the abstract expression class. This makes it easy to add more expression types. In addition, the representation as an abstract syntax tree does not require complex parsing and makes evaluation easy.

As these trees are part of the game world, they need to be serialized and de-serialized in order to write and read them from disc. To cover a large number of structurally different trees, Korat, TestEra and JCute are used to generate them. Each of these generated trees is serialized, de-serialized again and then compared to the original tree. This way we can test the serialization algorithm for a large number of different trees.

In this experiment, we compare the performance for generating these trees (the time for the serialization is not measured) for all three testing frameworks: Korat, TestEra and JCute. As TestEra does not support inheritance, we only use trees made of additions. In the JCute section, a binary search tree is generated, because JCute needs to create the tree via its API and therefore requires an insert method as explained in section 4.1.3.

5.3.1 Korat

Korat uses a finitization and a `repOK` method as a specification to generated non isomorphic test cases. Figure 5.2 shows several performance measurements of Korat for generating addition trees. The logarithmic y axis shows the required time in seconds. As before we used trees of different sizes: The number of used nodes are shown on the x axis. The different plots are explained throughout the following paragraphs.

When executing Korat, one can distinguish two phases: In the first phase, all classes are loaded and instrumented during the execution of the finitization method. For any given data-structure, the instrumentation takes a constant amount of time. The reason for this is that the number of classes to be instrumented does not change. In fig. 5.2 the

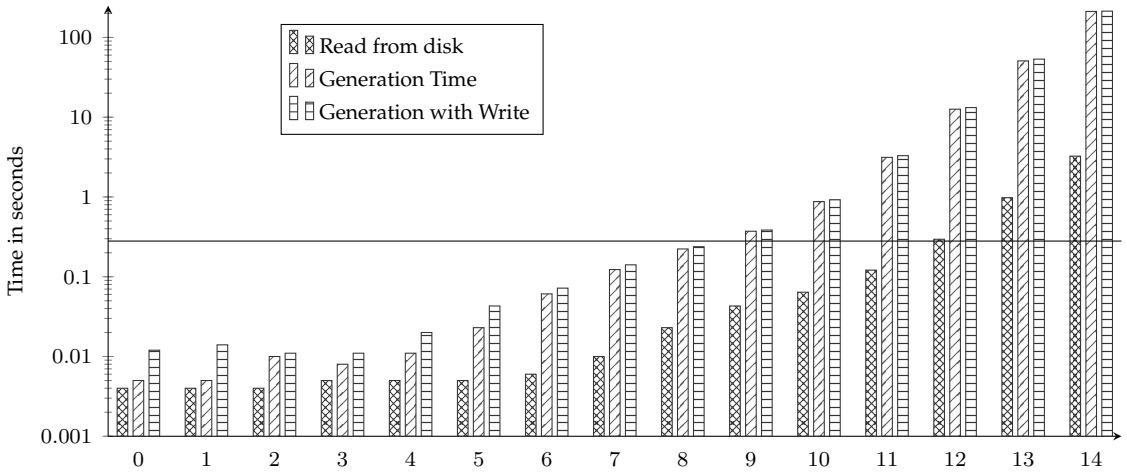


Figure 5.2: Analysis of Korat's runtime.

instrumentation phase is shown for the addition tree example as a horizontal black line at about 300 ms. Many projects aim at having a very fast running unit test suite, which can be executed in a matter of seconds before each commit. This makes it difficult to use Korat for these kinds of tests. Korat is better suited for long running tests, like regression or smoke tests on a server's setup.

During the second phase, Korat iterates over all elements of the search space, checks their validity using the `repOK` method and invokes the user's test function for valid elements. With a growing number of nodes, the number of possible structures grows as well and therefore iterating through the state space becomes more time-consuming. In the digram, the plot is named "Generation Time" and depicted by bars with diagonal lines. Observe how for small trees the instrumentation during the first phase is much slower than the generation itself. At the break-even point, at about 9 nodes, Korat already generates 6918 trees in about 600 ms. As the number of binary trees is growing very fast, the time required for the instrumentation becomes negligible in comparison to the overall time.

Korat offers utility functions to read and write candidate vectors from and to disk. Using these, it is easy to extend Korat to be able to write all valid candidate vectors during a normal execution to disk. By adding a special execution method, where the vectors are read from disk instead of being generated, one can significantly improve the performance of the state space exploration. The bars with the horizontal lines in fig. 5.2 show the required time for a normal state space exploration including the serialization of all valid candidate vectors to disk. One can see that, while it is slower, it does not worsen the performance much. The introduced overhead is repaid when reading the candidate vectors back.

The bars with grid pattern show the required time for reading candidate vectors from disc and passing them to the user's test function. It is clear to see that this method is a lot faster. There are a few reasons for that: First, Korat does not need to perform any compu-

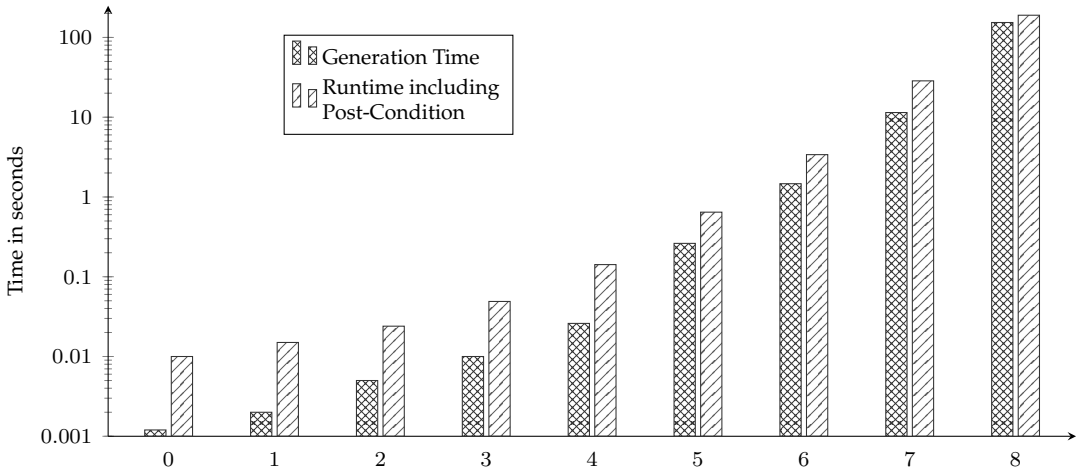


Figure 5.3: Analysis of TestEra's runtime.

tation for the iteration algorithm, the pruning filter or isomorphism filter. In addition, the file on disk contains only valid candidate vectors and thus Korat does not need to call the `repOK` method and automatically skips over all invalid candidates. As shown in fig. 2.6, most of the explored instances of a state space are invalid. Using the normal state space exploration algorithm, each candidate vector needs to be transformed into an object graph using Java's reflection API and checked by the `repOK` method.

5.3.2 TestEra

When using TestEra, the developer writes a specification for the structure to be generated. As described in section 3.1, there are several steps required to generated test input using this specification, but only two are interesting from a performance perspective: First, the generation of valid instances by the Alloy Analyzer and second, the execution of the test cases including the evaluation of the post-condition by the Alloy Analyzer. The other steps are only rather simple transformations between instances in Alloy and in Java.

Figure 5.3 shows a plot of the two interesting phases. As in the previous diagram, the y axis shows the time in seconds and the x axis the number of nodes. The bars with the grid pattern, labeled "Generation Time" show the time required for the Alloy Analyzer to generate the test cases from the specification[2]. The other bars, which are displayed with diagonal lines, show the time required to actually execute all JUnit tests. This includes the validation of the post-condition.

There are a few things to observe. First, the generation time (grid pattern bars) is higher than in Korat and it only scales to 8 nodes, within reasonable time constraints. Second,

[2]This also includes the time for the transformation from an Alloy instance into a Java object and also the time serializing them as JUnit test cases.

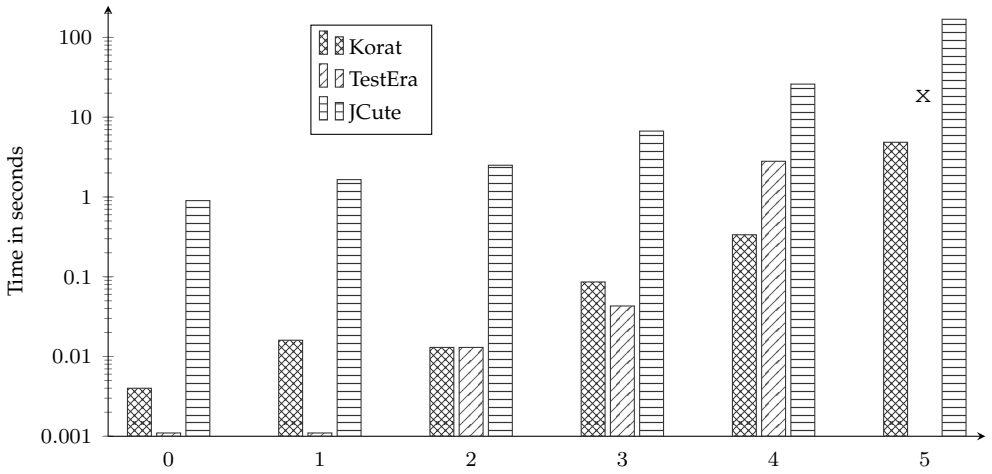


Figure 5.4: JCute's performance in comparison with Korat and TestEra.

the execution takes about the some amount of time for a larger number of nodes as the generation. This is due to the evaluation of the post-condition. To do this, TestEra abstracts the state before and after the execution of the method under test into an Alloy model. This model is required for checking the post condition. Even though the post condition was simply `true`, the abstraction process dominated the execution time.

With the post-condition not being used in this experiment, the abstraction of the pre- and post-state is also not required. When disabling it[3], the execution time of the test cases goes down to less than 1 ms and about 23 ms for 8 nodes. The resulting code does nothing else but creating objects. It is noteworthy, that Korat's second phase (the generation phase) needs roughly the same amount of time for the execution when reading from disk. Korat could skip the instrumentation when reading from disk or also generate code for creating the objects to get rid of the instrumentation overhead. This would allow Korat to be as fast as TestEra when running the tests multiple times.

5.3.3 JCute

JCute, as a concolic testing tool, is not specifically designed for generating data-structures. Its strength lies in finding specific inputs for a method in order to explore all possible paths through it. This feature can also be used for generating data-structures by repeatedly inserting entries as explained in section 4.1. When doing so, JCute tries to find different paths through the insert method and thereby creates structurally different data-structures.

The addition tree used in this experiment does not have an insert method. As there is no ordering criteria on the nodes, there is also no senseful way of writing one. Therefore, a binary search trees (as introduced in section 4.1.3) is used to analyze JCute and compare it

[3]TestEra does not provide an option to do that, but one can simple delete the code from the JUnit tests.

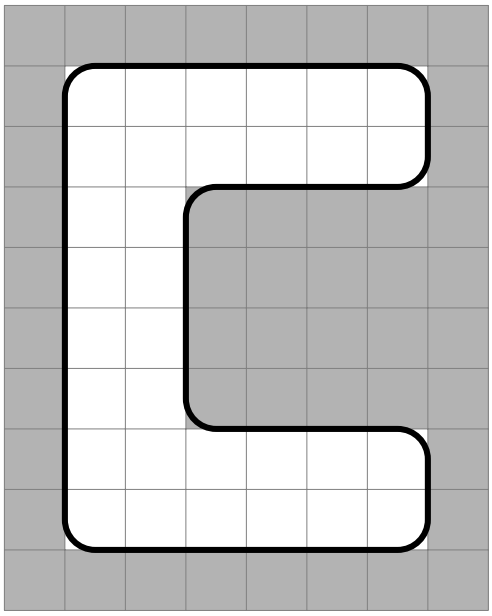


Figure 5.5: The collision map of a region in *Chelone*.

to Korat and TestEra. Figure 5.4 shows the performance for all three tools when generating a binary search tree. A logarithmic y axis is used to show the time in seconds and an x axis is used for the number of nodes.

Even on the logarithmic scale of the diagram, there is a clear difference between Korat and JCute: For 5 nodes, Korat is more than 30 times faster. TestEra does very well for tiny trees, but as the trees get larger (3 and 4 nodes), it becomes slower than Korat. On our system, TestEra is not able to generate binary search trees with up to 5 nodes. After a few minutes, TestEra crashes as indicated by the x in the diagram.

5.4 Experiment: Collision Detection

Chelone's game world, which can be explored by the player's character, is made up of regions. Each region is connected to a number of other regions. For example, consider a building in the game: The first floor is a region. By going to the stairs, the character is moved to the region for the second floor and when taking the exit door, the character is put into the outside region.

In each region, there are several objects with which the player can collide. In a building, these objects can be things like walls, cupboards, tables or computer controlled characters. To model the collision, two algorithms are used: The first is used for movable objects (like computer controlled characters or animals). As there are usually not too many of these in one region, *Chelone* uses an n^2 algorithm where each object is checked against all others.

The second collision system is more complicated and the focus of this experiment: For larger, immobile objects in the environment (like walls, cupboards or tables), *Chelone* uses a bitmap. An abstract sample of such a bitmap for the environment collision of a region is shown in fig. 5.5. Only the transparent fields can be accessed by the characters, while the gray fields represent areas, which can not be passed through. Observe, how this leads to rather sharp corners in the graphic. To solve this problem, *Chelone* uses a special algorithm for the collision, which we call *corner cutting* algorithm. Doing so provides a smooth in-game collision as shown by the black line.

The game world in *Chelone* is simulated using 30 small updates every second. In each of these game *ticks*, the player's character is moved by a small distance and the collision algorithms are used to check if the move is valid. Therefore, the collision algorithm has to be rather efficient. In addition, the correctness of the algorithm is essential, as a bug could cause the player's character to slip through a wall and be stuck behind it forever. As *Chelone's corner cutting* algorithm is rather complicated with about 100 lines of nested if-then-else constructs, it needs to be tested well.

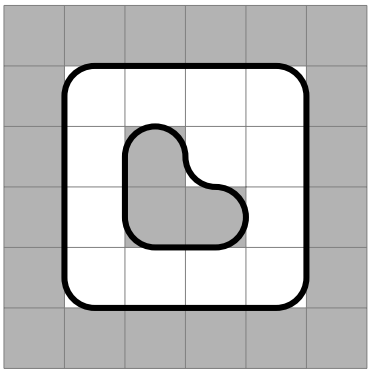
The position of the player's character is modeled using two floating point numbers. This means that a character can be positioned anywhere in the grid and not only at the center of the fields. Therefore, there is an almost infinite number of possible inputs for the *corner cutting* algorithm. This makes manual testing of the algorithm a time consuming task. To solve this, we try to automatically generate inputs for the *corner cutting* algorithm. We first look at the test code in Korat and JCute, after that we compare the results.

5.4.1 Korat

To test the *corner cutting* algorithm with Korat, we use one fixed collision bitmap and generate the position, movement direction and movement speed of the player's character with Korat. After that we let the game run until the character is unable to continue moving and validate that the character never occupies a field which is not walkable. From a game's perspective, the character is dropped at a random position and walks into a random direction until it hits a wall. The nice thing about this test is that it is easy to validate the correctness: The algorithm runs correctly for the given input if the character is not able to reach an invalid field and the moved distance is never greater than the movement speed.

The used collision bitmap is visualized in fig. 5.6, it is represented as a one-dimensional boolean array in Java. Korat is instructed to generate all floating point values between `1.0f` and `4.0f` with a sample rate of `0.2f` for the `x` and `y` component of the starting position. For the direction, Korat generates all angles, which are a multiple of ten, between `0` and `360` degrees. The speed is measured in fields per *tick* and is either `0.1f`, `0.2f` or `0.3f` in this experiment.

In order to not only test the *corner cutting* algorithm but also the game world, this test could be extended to use the collision bitmaps of the in-game regions instead of an artificial collision bitmap. This would have the advantage that it not only validates the *corner cutting*



```
1  if (y <= 0 || y >= 5
2   || x <= 0 || x >= 5)
3     return false;
4  if (y == 2 && x == 2)
5     return false;
6  if (y == 2 && x == 3)
7     return false;
8  if (y == 3 && x == 3)
9     return false;
10 return true;
```

Figure 5.6: Control flow graph of a simple function.

algorithm, but also the collision bitmap used in the game[4].

5.4.2 JCute

We also implemented a test for the *corner cutting* algorithm using JCute. The branch intensive nature of the *corner cutting* algorithm appears like an ideal candidate for JCute. Getting the test to run, however, took a lot more work than with Korat. When using JCute to generate values for a test, one needs to play a few tricks:

- **General Setup Modifications**
  As the performance experiments in section 5.3 have shown, JCute is slower than Korat. Therefore, the sampling rate of the direction angle needs to be reduced. In the setup for the JCute test, only three different angles are used: $0\,\mathrm{deg}$, $45\,\mathrm{deg}$ and $90\,\mathrm{deg}$. The position vector is generated using two `float` values obtained by the `Cute.input.Float()` function. The movement speed is constant.

  The test first generates the position and checks if it's a valid one (i.e. the field is walkable). After that, the direction is generated and one iteration of the movement algorithm is executed. Unlike in Korat, it is not possible to execute multiple iterations of the algorithm as the resulting symbolic state would be too large to handle it in a timely matter. Even with only one iteration, JCute takes about a minute to generate the test cases. Note that, after the test cases are generated, we remove this restriction and also let the game iterate until the character hits a wall.

- **Native Methods**
  When first running the test, JCute did not find many paths through the `corner cutting` algorithm code. The reason for this is that the algorithm uses many of Java's math utility functions. Many of these functions are implemented in native

[4]There are a few structures which break the *corner cutting* algorithm. Therefore, the level designer should never generate a collision bitmap with such a structure.

code, making it impossible for JCute to track the values. To solve this, we implemented our own math library in pure Java code.

Note that JCute is not able to track all variables symbolically, as there are many non-linear arithmetic operations (like square root or length of a vector) involved for the geometric calculations in the collision detection algorithm. But, as already claimed by the ordinal authors, this restriction rarely has practical implications: As shown in the result section, JCute is able to reach an optimal coverage.

- **Collision Bitmap Representation**
  Another issue is the representation of the collision information as a bitmap: When looking up a value in an array, there is an implicit branch based on the position of the lookup: Different lookup positions give different outputs. JCute does not include this branch in its symbolic tracking. This circumstance can be easily observed in the following example:

```
1 public static void main(String[] args) {
2    int[] arr = new int[]{6, 8};
3    int a = Cute.input.Integer();
4    Cute.Assert(arr[a] == 6);
5 }
```

  JCute only generates one test case, because it does not treat the array lookup as a branch. This is a problem for testing the *corner cutting* algorithm, as there are several lookups in the two-dimensional array representing the collision bitmap. To solve this, we remove the array and use a series of if statements to encode the same information. This allows JCute to track the branches correctly. The trick is shown in fig. 5.6: On the left hand side, the bitmap is shown and the corresponding code is shown on the right. Observe how both represent the same information.

5.4.3 Results

When comparing the two implementations, Korat has a clear advantage: It does not require the developer to change large portions of the code under test in order to generate the test cases. To make these required changes temporary, the developer could change the code, use JCute to generate the unit test, undo the changes and then use the generated JUnit tests on the original code. However, this process needs to be repeated every time the algorithm changes in order to be able to rerun JCute.

Both tools provide the same coverage. In the described setup, they explore all branches of the *corner cutting* algorithm, reaching a line coverage of 90%. The only lines missed are unreachable branches, which throw instances of the `UnreachableExceptoin` class.

In terms of performance, JCute has a clear advantage over Korat. After creating the 17 JUnit test cases in about 1 minute, executing them takes about 10 ms. Running Korat, however, takes about 590 ms in total. 90 ms are used to instrument the code and create the finitization. The remaining 500 ms are needed for the generation and the execution of

the test cases. These numbers can be optimized by using Korat's feature to read candidate vectors from file instead of generating them on the fly. This lowers the generation time to about 380 ms. In addition, the sampling rate could be reduced to lower the generation time further to about 50 ms, while maintaining the same coverage. When doing so, Korat creates 45 JUnit test cases. Even when rewriting Korat to create JUnit test cases like JCute, there is still a larger number of test cases which need to be run, because JCute only creates the ones which lead to different execution paths.

This experiment shows that automated testing tools can generate good test suites for algorithms, where the correctness can be easily validated. In addition, it shows that in this scenario the precise exploration of different branches by the concolic testing framework is able to achieve the same coverage with less test cases, which leads to a better performance. However, assuming Korat could also generate JUnit test cases, the performance difference would become fairly small.

5.5 Experiment: Item Bonus Calculation

This experiment shows one more example for how to use Korat for automated input generation. The key point of finding places to apply Korat is that there needs to be a semi-complex structure to generate, an algorithm to use on this structure and a simple way to validate the result. In the first two experiments, there was an algorithm to test, where the result was easy to validate. This is not always the case. Assume, for example, the generation of an entire game world: What would be the validation criteria? To validate an algorithm is often easier if it operates on a structure which has several invariants, as these can be used as a success criteria.

In this experiment we test the bonus calculation engine of *Chelone*: Each character in the game has several attributes like health, offensive and defensive abilities or magic skills. These attributes can be enhanced by wearing special armors or weapons. A sword, for instance, might increase the character's offensive ability. In addition, each character has a level which defines the base value for these attributes. Each attribute is represented as an integer value to the player. There are two kinds of bonuses: One is simply added to the base value and the other one multiplies the base value by a factor.

Calculating the value after all bonuses have been applied (i.e. the so-called effective value) can be done using the simple formula:

```
effectiveValue = baseValue * multiplicativeBonus + additiveBonus;
```

It gets more complicated for attributes which have a maximum and current value, like health for instance. Each character has a maximum health value and a current one. The current health is decreased whenever a monster hits the character and it can be increased by drinking potions or using healing spells. The issue is scaling it whenever the bonus changes. For instance, assume a character which has 100 max health and also 100 current health. Now this character equips an armor, which grants 20% additional health. Therefore the maximum health goes up to 120 points.

The issue is the current value: In *Chelone*, the current health value is also increased to 120, because otherwise the character would effectively be injured, as it would have only 83% health remaining. Now let's assume the character gets hit a few times and the current health drops to 5 points. What should happen if the player removes the armor again? Obviously, the character shouldn't die. Therefore, assume we reduce it down to 1 health point. What should happen when the player equips the armor again. Should it be scaled by the bonus factor, leading to a current health of 1.2, or should the absolute bonus be added, leading to 25 health?

These implementation details are not solved yet. To figure them out, it is helpful to test them with a lot of different equipment items. This is were Korat comes into play: We use Korat's ability to generate large amounts of items and use these to dress and undress a character. Using the following simple rules we can validate the bonus calculation algorithm:

- A character should not die when changing the equipment.
- Assuming a character without any equipment: Equipping and removing a piece of equipment should not change the current or maximum value for any attribute.
- Assuming a character which has one equipment slot equipped with a certain item: Removing this item and re-equipping it should not change the current or maximum value for any attribute.

Having this testing system, powered by Korat, in place, it is easy to check if a new algorithm still meets the specified criteria. Also if it turns out that these criteria can not be fulfilled, they can easily be changed. This experiment shows an ideal use case for Korat: There is a semi-complex structure to be generated (a set of items with bonuses), and there is an algorithm, where it is easily possible to specify its post condition. Manually specifying these items would take a lot of time. Korat is able to generate about 8.000 items in roughly 600 ms. This provides a large pool of testing equipment.

This experiment is also an example for a use case where the new finitization (see section 2.3.6) is required: To implement the bonuses, *Chelone* uses an expression tree (as introduced in section 5.3) for the additive and multiplicative bonus. Both fields have the same class (i.e. `Expression`), but one needs integer values and the other floating points. The new finitization concepts make it possible to use different class domains for both fields and thereby have different values associated.

5.6 Experiment: Input Queue Synchronization

This experiment shows an example of how to use JCute's feature to test multi-threaded code and evaluate its results. *Chelone* uses two threads: One is the system's thread, which is used to draw the user interface and to handle user input events. The other one is the

```
1  public class UserInputQueue {
2
3     private List<UserCommand> activeQueue = new ArrayList<>();
4     private List<UserCommand> passiveQueue = new ArrayList<>();
5
6     /// Called by GUI
7     public synchronized void addCommand(UserCommand command) {
8        if(!activeQueue.contains(command))
9           activeQueue.add(command);
10    }
11
12    /// Called by game
13    public void processInput() {
14       swapQueues();
15       for (UserCommand userCommand : passiveQueue)
16          userCommand.execute(); // Could potentially block
17       passiveQueue.clear();
18    }
19
20    private synchronized void swapQueues() { /* ... */ }
21 }
```

Listing 5.1: User input handling in *Chelone*.

game logic thread. It is responsible to update the game state 30 times per second. As multi-threaded code is usually not easy to write and even harder to debug, *Chelone* tries to bundle the synchronization code in a few classes. To achieve this, message passing is used to send input events from the user interface thread to the game logic thread.

This mechanic is implemented via a queue. The code is shown in listing 5.1. There are two important methods, which are called by the two threads:

- The `addCommand` method in line 7 is called by the interface thread whenever user input needs to be passed to the game thread. This input event is recorded as a `Command` object and added to the `activeQueue`.
- To process the `Command` objects on the game thread, the `processInput` method is used. It first swaps the `activeQueue` and `passiveQueue` queue. This is done in the synchronized `swapQueues` method. In Java, the execution of two methods declared with the `synchronized` keyword is mutually exclusive within the context of one object. Hence, the code of `addCommand` and `swapQueues` can not be executed at the same time and is therefore serialized.

  After the queues have been swapped, the game thread can process the events in the `passiveQueue` without any need for synchronization, as the interface thread is only adding `Command` objects to the `activeQueue`. By swapping the queues, the `processInput` does not need to block the `addCommand` method while processing the `Command` objects. This non-blocking processing is necessary, because some of

the code encapsulated in the `Command` objects require waiting for the user interface thread.

To test this code via JCute, two threads are used. The first thread, which represents the user interface thread, produces two `Command` objects and passes them to the game logic using the `addCommand` method. The second one simulates the game logic and calls the `processInput` method once.

JCute is able to find all three schedules, where the access order to the shared data-structure is different. In this case, the shared data-structures are the two queue objects. JCute is able to find all three different execution orders:

1. Add both `Command` objects and then process them.
2. Add one `Command` object, process it and then add the other one.
3. Execute the `processInput` method with an empty queue and then add both `Command` objects.

In order to explore these different scheduling orders, JCute instruments the code as explained in section 4.2.4 and adds additional mutexes to control the scheduling. Hence, JCute is not able to generate JUnit tests for these concurrency tests, as the instrumentation is required to make it work. Therefore it can only be run from within the JCute framework. Doing so takes about 3 to 4 seconds. Hence, when willing to set up an environment in which JCute can be run, this could be integrated into the build chain of a program. But even if not, this concurrency testing feature of JCute is an easy way to find concurrency issues in code.

6 Conclusions and Future Work

Software development projects can greatly benefit from automated test input generation. It lowers the amount of manual work and can increase the test coverage: As shown by the conducted experiments, it is possible to exhaustively explore all instances of a given data-structure up to a specified size. This allows the developer to test an algorithm for every small instance of the data-structure.

Throughout this thesis, two tools for specification-based testing and one for concolic testing were evaluated. This chapter first sums up the features of each tool individually and then provides a comparison between them in section 6.4. Each individual section describes the tool's limitations, possible extensions for future work and summarizes the experimental results.

6.1 Korat

Korat is a very promising tool for automated test input generation. Using only a finitization and a validation method (`repOK`), it is able to generate a huge number of inputs in little time. When exploring large search spaces, Korat optimizes the search by skipping over large portions of the search space. This is achieved by a pruning filter, which removes invalid portions of the search space, and an isomorphism filter, which skips over structurally equal object graphs.

The standard Java language is used to express the specifications. In addition, both parts of this specification – the finitization and validation method (`repOK`) – are imperative program snippets. For many developers, this offers a very natural way to specify an object graph, as it does not differ much from usual programming.

6.1.1 Limitations

From the three tools tested in this thesis, Korat is definitely the one on which the most time was spent. As described in section 2.3, we resolved several issues and bugs. This makes it possible to use Korat for a wider range of projects. Still, there are some open issues remaining which limit the scope in which Korat can be used. First, the instrumentation fails for inner classes and second, multidimensional arrays are not allowed in the finitization. While the second issue should not be very important, the first one is troubling and should be resolved to allow for a broader usage of Korat.

All resolved issues and remaining bugs in Korat show that it is very difficult to write a correct instrumentation module. There are a lot of different code patterns which need to

be instrumented and it is very difficult to test all of them during development.

6.1.2 Debugging with Korat

As described, Korat instruments the code to track accesses to the generated objects. While this allows to trim the search space, it introduces additional code, like the KoratArrays and getter methods at runtime. Using a debugger is therefore complicated, as the runtime generated code does not show up.

When testing with Korat, a large number of test inputs are used and for each one the user's test function is called. When generating larger data-structures, these tests can require some time to run. Therefore, Korat offers an option to output the current candidate vector. This feature allows the developer to re-run one specific candidate vector in case of an error. To accomplish this, the developer can pass the candidate vector directly to Korat's API and thereby re-run just one test input.

6.1.3 Future Work

As mentioned above, there are still a few open bugs in Korat which need to be resolved to make Korat usable for more projects. In addition, Korat needs some refactoring on its interface. This includes cleaner API functions, a better documentation and meaningful error messages. These features are required to help new users to get familiar with Korat. There are also some interesting extensions, besides bug fixes and refactoring, for Korat:

Currently, Korat exhaustively explores all valid instances for a given specification. In some cases it could be helpful to offer techniques like *all-pairs testing* to reduce the number of generated instances. This would allow a developer to use Korat in setups with lots of independent fields in the test structure.

As shown in the *collision detection* experiment in section 5.4, Korat is faster than JCute when generating test inputs, but JCute is able to create JUnit test cases for each generated test instance. Running these JUnit tests is a lot faster than generating the tests each time with Korat. Adding a feature to create JUnit tests for each valid candidate vector would allow Korat to compete with JCute in setups where the test input is generated once and run multiple times.

Another possible feature for Korat is to combine it with a coverage tool. After generating test cases and writing them to disk, either as candidate vectors or JUnit tests, these test cases are executed over and over again. Many of these test cases are redundant, as they take the same path through the test code. The idea is to track which lines or branches are covered during generation for each candidate vector and only keep those which improve the coverage. This feature would allow Korat to generate a similar number of test cases as JCute, while still offering the easy to use specification-based way of specifying the input structure.

6.2 TestEra

TestEra provides a novel approach for testing programs: Using the Alloy language and the Alloy Analyzer, one can automatically generate input for test cases and directly verify the method under test using pre- and post-conditions. To achieve this, TestEra supplies the framework, which connects the Alloy Analyzer with Java.

6.2.1 Limitations

When using TestEra in a real world project, one is likely to encounter several difficulties as explained in section 3.3. These issues prevent the developer from testing structures which use primitive data types other than boolean or integers. In addition, the lacking support for inheritance and nested classes requires the developer to write additional code to hide these features from TestEra. The newly written code has to be maintained and potentially contains bugs which nullify the test cases. Hence, TestEra can only be directly and efficiently used for small parts of the project.

6.2.2 The Alloy Language

The specifications in TestEra are written using Alloy, which is a first order relational language. In order for TestEra to be used in a project, the developers that are responsible for testing[1] have to learn this new language. Note that this might be more difficult for many programmers which only have a background in procedural programing languages and are not trained in formal ones.

But, if this initial barrier is taken, Alloy offers a very expressive way to model a method's behavior. In addition, the specification can be directly written inside the class to be tested. While this approach puts two languages in one file and also mixes testing code and production code, it has the advantage that the formal specification of a method is also a very precise way of documenting the method. Hence, one could skip the imprecise informal documentation.

Specifying valid instances of a class using the Alloy language is not a trivial process. To support the developer, TestEra offers a visualization of the generated structures via the Alloy visualization API. With this feature, a developer can work iteratively on the specification. Looking at the output, it is easy to find invalid instances and refine the specification to prevent these from being generated.

6.2.3 Development Environment

TestEra is integrated into the Eclipse IDE[2]. This allows the developer to use a graphical user interface to interact with it. There is no integration for other development environ-

[1] In modern, more agile, engineering teams, all developers are responsible for testing.
[2] The integration only works with the rather out of date version 3.6 Helios (2010) of Eclipse.

ments, forcing the developer to use TestEra's command line interface. However, integrating TestEra into other development environments should be rather easy, as there are only a few simple commands which have to be invoked to trigger the test case generation. In addition, the command line API also allows a developer team to integrate TestEra into their build tool chain.

TestEra works in two stages: First, the developer uses TestEra to generate the Alloy model and the test cases. Second, the generated test cases are run using JUnit. As shown in section 5.3, the first phase is rather slow, as it has to use the constraint solver within the Alloy Analyzer to generate the instances according to the specification. But in the second stage, one only has to run the JUnit test cases. Within the test case there is only plain Java code, which constructs the instances and therefore it can run pretty fast, assuming the post conditions are not used. This system works very well for typical unit tests: The tests are written (generated) once and executed within a regression test suite multiple times.

6.2.4 Future Work

As with most research projects, TestEra still has several shortcomings (as described in section 3.3). Compared to the other tools it is the one with the fewest features. To make use of TestEra in a real world project and to test not only data-structures, these issues would need to be resolved. Especially primitive data types and inheritance are required in almost every project.

TestEra was developed a few years before Korat at the same University. Judging from our experiments and experience using these tools, Korat seams to be the successor of TestEra. As both are specification based testing tools, they stand in direct competition towards each other. When starting a new project and deciding which tool to use, Korat is likely the better way to go: Due to the limitations of TestEra, Korat offers a wider range of use cases.

6.3 JCute

JCute brings the well known concolic testing framework – Cute – to the world of Java. When using JCute to test a method, it tries to explore all different execution paths through the method. For each different path found, JCute can create a JUnit test case. This allows an easy creation of test suites with a high coverage percentage.

6.3.1 Platform

JCute itself is written in Java and therefore can be used on any platform which offers a JVM. But JCute also depends on the lp_solve library, which is openly available under the GNU lesser general public license [6]. To make use of lp_solve, a custom wrapper library is required, which needs to be compiled by the tester. In larger projects which span multiple platforms, this could be an issue. One solution is offered in the form of a pre-compiled

wrapper including lp_solve. However, this binary distribution of JCute depends on some older 32 bit Linux libraries. Therefore it is only usable on older machines or inside virtual machines.

Once JCute is installed, it is very easy to get started using it: Its graphical user interface provides an easy way for running and evaluating the results of the created tests. In addition, JCute can also be used from the command line. This allows using it from within an automated build system, like a continuous integration server.

6.3.2 Features

Unlike in TestEra, the developer does not have to write a specification in a formal language. In JCute, the developer creates a data-structure simply by successively adding elements to it. This makes it a very natural process which is easy to use. However, as shown in section 5.4, it can be difficult to make JCute work with certain algorithms.

JCute explicitly supports concurrency. Using JCute, it is easy to explore various corner cases in multi threaded code. This is an especially useful feature, as multi threaded code is often difficult to test and understand. Therefore, having a tool for effectively testing it should prove very beneficial for the development team.

6.3.3 Future Work

JCute is the most stable of the three tools. We only encountered one problem during our experiments: JCute does not always create a JUnit test case for each of the explored paths. For example, in the binary tree test in section 4.1.3, JCute explores all 8 structurally different trees, but only creates test cases for 5 of them. This dramatically reduces the value of the created JUnit test suite, as the coverage is much lower.

Another feature for JCute would be an optional, more powerful constraint solver for non-linear expressions. While there were no problems in our experiments, it could be an issue for algorithms with more complex calculations. Therefore, having another solver as a fallback would be beneficial.

6.4 Overall

All three tools are written to test Java code and also are implemented in Java. JCute uses the lp_solve library, which is not available in Java. The others, Korat and TestEra, only have dependencies on Java libraries. This allows them to be used on any platform which offers a JVM.

All of the used tools have several shortcomings, as they are only research prototypes. While this is fine as a prove of concept, these issues have to be resolved to make them usable in real world projects. For most companies, it is not feasible to use an unstable tool for their testing or to fix bugs in the testing framework. Therefore, before any of the tools can be used in an industry project, they would need to become more stable.

Our development on Korat leads it into the right direction. By fixing bugs and extending the test suite, Korat is now a lot more stable and usable for a wider range of projects. To make it easier to get started, the tools would also need a better documentation and error reporting at runtime.

Given the outlined limitations and features, Korat is probably the best choice for most projects. As stability and reliability, especially for a testing framework, are a huge factor for any real world project, TestEra is not a good candidate in its current state. Compared to JCute, Korat offers an exhaustive exploration of the search space, no required external libraries and also a better performance for test case generation.

Bibliography

[1] Catalan Numbers. https://oeis.org/A000108.

[2] Eclipse IDE. https://eclipse.org/.

[3] GNU General Public License, Version 2. http://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html.

[4] Java SE7 TreeMap Reference. https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html.

[5] Java SE7 TreeSet Reference. https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html.

[6] LGPL: GNU Lesser General Public License 2.1. http://lpsolve.sourceforge.net/5.5/LGPL.htm.

[7] lp_solve. http://lpsolve.sourceforge.net.

[8] Thomas Ball. Abstraction-Guided Test Generation: A Case Study. Technical report, Technical Report MSR-TR-2003-86, Microsoft Research, 2003.

[9] Rudolf Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[10] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[11] Boris Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[12] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Softw. Eng. Notes*, 27(4):123–133, July 2002.

[13] Patrice Chalin, Joseph R Kiniry, Gary T Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal methods for components and objects*, pages 342–363. Springer, 2006.

[14] Shigeru Chiba. Javassist-a Reflection-based Programming Wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, page 174, 1998.

[15] Thomas H Cormen. *Introduction to Algorithms*. MIT press, 2009.

[16] Christoph Csallner and Yannis Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004.

[17] Christoph Csallner and Yannis Smaragdakis. Check 'N' Crash: Combining Static Checking and Testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[20] Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pages 8–21, Washington, DC, USA, 1978. IEEE Computer Society.

[21] Hans-Martin Hörcher. Improving Software Tests Using Z Specifications. In JonathanP. Bowen and MichaelG. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin Heidelberg, 1995.

[22] Daniel Jackson. Automating First-Order Relational Logic. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 130–139. ACM, 2000.

[23] Daniel Jackson. Alloy 3.0 Reference Manual. *Software Design Group*, 2004.

[24] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: The Alloy Constraint Analyzer. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 730–733. IEEE, 2000.

[25] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A Micromodularity Mechanism. *ACM SIGSOFT Software Engineering Notes*, 26(5):62–73, 2001.

[26] Gosling James, Joy Bill, Steele Guy, Bracha Gilad, and Buckley Alex. The Java Language Specification, Java SE 8 Edition. `https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html`, February 2015.

[27] Jeff Carollo James A. Whittaker, Jason Arbon. *How Google Tests Software*. Addison Wesley, 1 edition, 2012.

[28] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Massachusetts Institute of Technology, 2003.

[29] Eric Larson and Todd Austin. High coverage detection of input-related security faculties. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[30] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[31] Gary T Leavens and Yoonsik Cheon. Design by Contract with JML, 2006.

[32] Donald M. Leslie. Using Javadoc and XML to Produce API Reference Documentation. In *Proceedings of the 20th Annual International Conference on Computer Documentation*, SIGDOC '02, pages 104–109, New York, NY, USA, 2002. ACM.

[33] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Massachusetts Institute of Technology, 2005.

[34] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An Evaluation of Exhaustive Testing for Data Structures. Technical report, Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, 2003.

[35] Darko Marinov and Sarfraz Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 22–31. IEEE, 2001.

[36] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[37] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.

[38] A. Jefferson Offutt and J. Huffman Hayes. A Semantic Model of Program Faults. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 195–200, New York, NY, USA, 1996. ACM.

[39] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *CAV*, pages 419–423, 2006.

[40] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[41] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[42] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[43] Alexis M Tourapis, Oscar CL Au, and Ming L Liou. Predictive Motion Vector Field Adaptive Search Technique (PMVFAST): Enhancing Block-based Motion Estimation. In *Photonics West 2001-Electronic Imaging*, pages 883–892. International Society for Optics and Photonics, 2000.

[44] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.