HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Laboratory of Software Technology

Petri Ihantola

Automatic test data generation for programming exercises with symbolic execution and Java PathFinder

Master's Thesis.

Espoo, May 30, 2006

Supervisor: Professor Lauri Malmi

Instructor: Docent Ari Korhonen

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Petri Ihantola	
Name of the thesis:	Automatic test data generation for programmign exercises with symbolic execution and Java PathFinder	
Date:	May 30, 2006	**Number of pages:** 77 + 6
Department:	Department of Computer Science and Engineering	**Professorship:** T-106
Supervisor:	Prof. Lauri Malmi	
Instructor:	Docent Ari Korhonen	

Besides of software industry applications, a typical example where automated verification techniques are applied is the automatic assessment of programming exercises The assessment is typically based on testing. Although automatic assessment frameworks execute tests and evaluate test results automatically, test data generation is not automated.

PathFinder is a software model checker, which can also be used for automatic test data generation. In this thesis, we will study how the tool can be modified for the needs of automatic assessment. Practical problems considered are: How to derive test data directly from students' programs? How to demonstrate test data (and thereby test cases) automatically for students?

Most test data generation tools supporting linked data structures require annotations to the original program or use specialized specification languages. Thus, Creating test data directly from students' programs is problematic. Test data generation with Java PathFinder is also based on annotation. In this thesis we will introduce different automated approaches to replace the manual annotation.

Test data demonstration presented in this work is based on partially initialized object graphs and symbolic execution. The idea is to provide symbolic object graphs for students. Each symbolic object graph can be instantiated into (several) concrete test data so that all the inputs derived from the same symbolic structure lead into identical execution paths.

Keywords: testing, unit testing, test data generation, symbolic execution, model checking, programming exercises, programming education

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Petri Ihantola	
Työn nimi:	Testisyötteiden automaattinen luominen ohjelmointiharjoituksiin symbolisen suorituksen ja Java PathFinderin avulla	
Päivämäärä:	30.05.2006	**Sivuja:** 77 + 6
Osasto:	Tietotekniikan osasto	**Professuuri:** T-106
Työn valvoja:	Professori Lauri Malmi	
Työn ohjaaja:	Dosentti Ari Korhonen	

Ohjelmien automaattista verifiointia käytetään paitsi ohjelmistoteollisuudessa, myös tietotekniikan opetuksessa ja erityisesti ohjelmointitehtävien automaattisessa arvioinnissa. Testaus on automaattisessa arvioinnissa tyyppillisimmin käytetty verifiointimenetelmä. Vaikka automaattisessa arvioinnissa käytetyt työkalut automatisoivatkin testauksen eri vaiheista testien suorittamisen ja testitulosten arvioinnin, testisyötteiden suunnittelu tehdään edelleen käsin.

Tässä työssä tarkastellaan erästä testisyötteiden automaattiseen suunnitteluun soveltuvaa työkalua (Java PathFinder mallintarkistin) ja sen muokkaamista ohjelmointiharjoitusten erityistarpeisiin. Työssä keskitytään kahteen konkreettiseen ongelmaan: Kuinka testisyötteitä voidaan johtaa suoraan opiskelijoiden ohjelmista? Kuinka suunniteltuja testisyötteitä ja testitapauksia voidaan automaattisesti havainnollistaa opiskelijoille?

Testien luonti suoraan opiskelijoiden ohjelmista on ongelmallista. Useimmat viittauksia sisältäviä tietorakenteita käsittelevistä testisyötteiden suunnittelujärjestelmistä käyttävät syötteenään työkalun tarpeisiin annotoitua versiota alkupeäisestä ohjelmasta tai erityisellä määrittelykielellä laadittua ohjelmaa. Java PathFinderin käyttö testisyötteiden suunnittelussa perustuu annotointiin. Työssä esitetään erilaisia lähestymistapoja, joiden avulla manuaalinen annotaatio voidaan välttää.

Työssä esitelty testisyötteiden havainnollistamismenetelmä perustuu osittain instantioituihin tietorakenteisiin (oliograafeihin) ja symboliseen suoritukseen. Havainnollistamisen perusajatuksena on tarjota opiskelijoille symbolisia tietorakenteita siten, että kaikki yhdestä tietorakenteesta instantioidut testisyötteet johtavat identtisiin suorituspolkuihin testauksen lähtökohtana käytetyssä ohjelmassa.

Avainsanat: testaaminen, yksikkötestaus, testisyötteiden suunnittelu, symbolinen suoritus, mallintarkistus, ohjelmointiharjoitukset, ohjelmoinnin opetus

Acknowledgements

This Master's thesis has been done for the Laboratory of Software Technology.

I want to thank my instructor Ari Korhonen and supervisor Lauri Malmi.

Especially I want to thank Willem Visser, who provided the symbolic execution library of JPF for this study.

And a special thanks goes to Maija and Taavetti for being special.

Otaniemi, May 30, 2006

Petri Ihantola

Contents

Chapter 1

Introduction

Software errors (*i.e.* bugs) can cause huge financial losses (*e.g.* Ariane 5, flight 501 crash [10]) or even cost human lives (*e.g.* Therac-25 radiation therapy machine failure [43]). Therefore it is not a surprise that *software verification* is an important part of any software development process. The goal of verification is to evaluate the conformance of a software against the corresponding functional specifications.

Verification is typically divided into two different approaches: *formal verification* and *software testing*. Perhaps the most fundamental difference between these two approaches was pointed out by Dijkstra [20, page 8] in 1972: "testing can be used to show the presence of bugs, but never their absence.". On the other hand, formal verification can also address the absence of bugs. Yet, when discussing automated verification (testing or formal verification), we should remember that there are properties that cannot always be checked, no matter what kind of verification techniques are used. The classical example is that one cannot automatically verify if the execution of a program finally halts, *i.e.* Turing halting problem [59, 60].[1]

Besides software industry applications, typical examples where automated verification techniques are applied are numerous assessment systems widely used in computer science (CS) education (*e.g.* ACE [55], TRAKLA2 [40], and PILOT [13]) – especially in systems used for automatic assessment of programming exercises (*e.g.* ASSYST [31], Ceilidh [11], JEWL [24], and SchemeRobo [54]). Automatic assessment of programming exercises is typically based on testing approach and seldom on deducting the functional behavior directly from the source code (such as static analysis in [58]). In addition, it is possible to verify features that are not directly

[1] In practice, formal verification, and especially model checking is sometimes capable to show if a program has an infinite execution path or not.

related to the functionality. For example, a system called Style++ [3] evaluates the programming style of students C++ programs.

The focus of this work is on testing, not on formal assessment. Later on, when referring to *automatic assessment*, we mean automatic verification of programming exercises by using the testing approach, not formal assessment. Programming style related topics are also ruled out.

Testing can be defined using different words, but the common idea behind most definitions is that testing is a process of increasing trust about the correctness of a program by executing it with different inputs. Thus, the first thing to do is to select a representative set of inputs. Input for a single test is called *test data* or *test input*. After test data selection, the correctness of the behavior of the program is evaluated. That is to say, a *test oracle* is applied. The test data and the corresponding oracle together are called a *test case*. Finally, the test data of a logical group of tests together are called a *test set*.

Test data selection and test set generation can be extremely labor intensive. Therefore, automated methods for the process have been intensively studied for over decades (*e.g.* [16, 32, 53]). In this work, we will address special problems of using such techniques to create test data for automatic assessment.

1.1 The problem

Manual test data selection for programming exercises is tedious and error prone. One can easily leave relevant test data out from the test set. Another problem is that good test coverage for the specification does not guarantee good test coverage for the students' programs. This is a problem because it is impossible to give feedback from parts of programs that are not executed. Moreover, the lack of feedback can be a misleading feedback. If the parts of programs that are not assessed do not work correctly, the lack of feedback could promote thinking that the program is correct. Furthermore, this creates and feeds students' misconceptions [56], which is harmful for learning.

The following formalism is used to explain test data generation. Automatic assessment, as testing in general, starts from a specification ($\mathcal{S}$). A teacher defines a test set ($\mathcal{T}$) to cover different aspects of the specification $\mathcal{S}$. In addition, a teacher can figure out typical misconceptions and design test data to reveal programs following such faulty mental models [47]. The process is often completely manual. A student

Figure 1.1: Different approaches for code driven test data generation: the traditional approach based on a specification on left and our approach based on the specification and the program together on right.

implements a candidate program ($\mathcal{P}$). Finally, the oracle is based on an automatic comparison of the candidate program and the specification with test data in the selected test set (*i.e.* conclude if $\mathcal{P}(\mathcal{T}) = \mathcal{S}(\mathcal{T})$). For each test data in the test set, the feedback shows that either $\mathcal{P}(data)$ fails or works correctly. In addition, the teacher can include a textual demonstration[2] to each test data. In such cases, textual demonstrations are also part of the feedback.

To provide better test adequacy, the candidate program should also be included when the test data is produced. The difference between such test data generation and the test data generation traditionally used in automatic assessment is illustrated in Figure 1.1. In the traditional approach, $\mathcal{T}$ is derived only from $\mathcal{S}$. In the proposed approach $\mathcal{T}$ is derived from both $\mathcal{S}$ and $\mathcal{P}$.

Automatic test data generation is an essential requirement to derive $\mathcal{T}$ also from $\mathcal{P}$. The reason for this is the *automatic* assessment. Manual test data generation separately for each submission leads into *manual* or *semiautomatic* assessment. However, automatic test data generation from students' programs is not straightforward. The reason for problems is that most test data generation tools working with references are either 1) using some specification languages (*e.g.* AsmLT [8]), or 2) require annotation of the code (*e.g.* Java PathFinder (JPF) [64]).

Yet another problem is to demonstrate why certain test data are included to the test set. When test data are derived from the specification, the demonstration can be done manually. However, it is time consuming to find out the role of automatically generated test data (*i.e.* it is difficult to demonstrate why a certain test is executed). Furthermore, in test data derived from students' programs, the manual demonstration is not possible at all. The task is impossible because test data is generated on-the-fly, whenever a student submits a solution.

Symbolic execution [37] demonstrates program behavior on a higher abstraction level

[2]The word *demonstrate* is used throughout the thesis to describe the process of describing and explaining test data, and thereby test cases, for students. There are other possibilities (*e.g.* illustrate or visualize) but we have selected to be consistent and only one term is used.

than simple traces. On the other hand, some test data generation tools (*e.g.* Java Pathfinder [64] and Symstra [67]) are strongly based on symbolic execution. We believe that such tools can be extended, not only to create test data, but to demonstrate it. Our special interest is on JPF, as it is publicly available, whereas Symstra is not. Thus, the research question of this thesis can be formulated as follows:

*How can we automatically derive and demonstrate test data directly from implementations (*i.e. *programs) with symbolic execution and JPF?*

The formulation does not limit the traditional definition of specification based test data generation – it only extends the traditional definition. As will be explained in Section 1.4, we understand specifications to be programs[3]. Later on, the word *program* is used for *candidate programs* (*i.e.* programs implemented by students) and *specifications* (*i.e.* model solutions implemented by a teacher) together.

We underline the difference between specifications and assignments. Specification is a functional model. It is a model solution for the programming exercise, whereas the assignment is an abstract description about the goals of the programming exercise. The assignments are for students whereas specifications are for automatic assessment. Both are needed in automatically assessed programming exercises.

Even though it is important to create adequate tests to reveal possible bugs, it is equally important to locate these bugs. Most automated test data/case generators are designed for professionals, who have good debugging skills (*i.e.* skills to locate and correct errors). Unfortunately, the majority of novice computer science students are not good at debugging [1].

As a summary: 1) To provide better test adequacy and therefore more comprehensive feedback, tests should be derived from both, specification and candidate program instead of specification only. 2) Automatic test data generation is essentially needed, but the new problem is how to demonstrate automatically produced test data for students. 3) In this work, we are looking for a solution for these problems with symbolic execution and JPF.

[3] "A program specification is the definition of what a computer program is expected to do. It can be informal, in which case it can be considered as a blueprint or user manual from a developer point of view, or formal, in which case it has a definite meaning defined in mathematical or programmatic terms." `http://en.wikipedia.org/wiki/Program_specification` [Visited May 26, 2006]. In this work we have selected the latter (*i.e.* formal) interpretation for the word *specification*.

1.2 A solution

In this work, we present a novel idea to automatically extract abstract test demonstrations from the test data. A technique to automatically generate test data with integer constraints and object graphs is demonstrated. Conceptually, in our approach, the outcome of automatic test data generation is not only a test set, but a set of *test patterns*. Each test pattern defines test data that are somehow similar. Test pattern is a kind of opposite to test set because the latter contains different test data in order to provide good test coverage. A possible grouping criteria for test patterns is that the execution paths in the program are identical. In detail, a test pattern consists of a single *test schema* and possibly several test data derived from the schema. All the test data in the same test pattern are derived from the schema of the pattern. Finally, the test set is obtained by selecting arbitrary test data from each test pattern.

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

The schemas will be used to demonstrate tests on a higher abstraction level when compared to the actual test data. To understand the use of schema in the feedback, let us consider test schema $\mathbf{s}$ and test data $\mathbf{t}$ derived from $\mathbf{s}$. Instead of exact feedback saying $\mathcal{P}(\mathbf{t})$ fails (or works correctly), we will provide abstract feedback like "$\mathcal{P}(\mathbf{s})$ fails (or works correctly)". However, the oracle of the automatic assessment is based on investigating $\mathcal{S}(\mathbf{t}) = \mathcal{P}(\mathbf{t})$, as in the traditional approach. Figure 1.2 illustrates this process and the related terminology.

This work is strongly based on techniques and tools developed by Willem Visser, Corina Păsăreanu, Sarfraz Khurshid, and others [5, 12, 36, 51, 62, 64]. The main contribution in this work aims to describe how the test data generation with JPF can be developed towards the needs of automatic assessment:

1. to provide more comprehensive feedback by deriving the test data from candidate programs instead of annotated specifications only;

2. not only to create, but also demonstrate the test data produced.

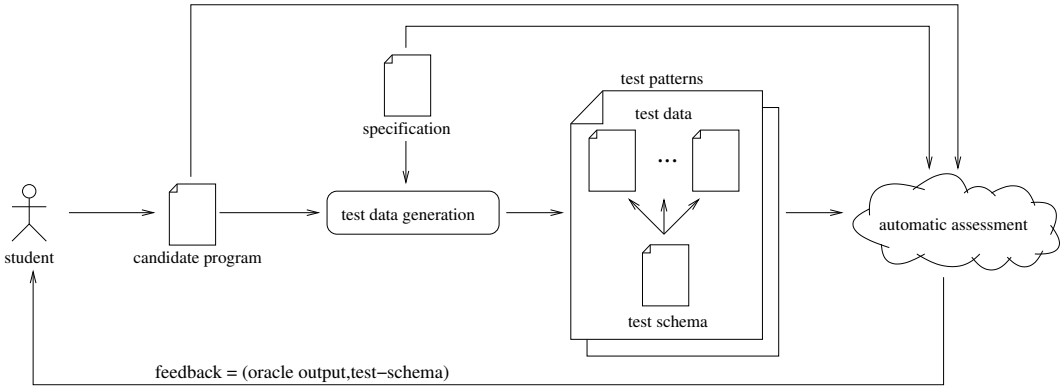

Figure 1.2: The process of creating feedback for students and some related terminology. Feedback from a single test is a pair including the test schema from where the test data is derived and the oracle result with the test data.

1.3 A simple example

Consider a simple programming assignment, where the objective is to calculate distance between two integers. Program 1.1 is an example specification for the task. In the traditional approach, a teacher would figure out that tests where $a < b$ and $a > b$ are relevant. The actual test data could be something like $\{a = 2, b = 4\}$ and $\{a = 4, b = 2\}$. The constraints (or similar textual demonstrations) are manually linked to the test data. For example, the feedback from Program 1.2 might be something like: "Your program passes tests where $a > b$, but fails tests where $a < b$."

```
1 int distance( int a, int b ) {
2   if ( a < b )
3     return b - a;
4   else
5     return a - b;
6 }
```

Program 1.1: A specification calculating the distance between two points.

Currently JPF can be used to generate two test data from the example specification. The test data could be the same as generated by the teacher: $\{a = 2, b = 4\}$ and $\{a = 4, b = 2\}$. However, without additional work from the teacher, the feedback from the faulty program would be "program work correctly where $a = 4, b = 2$ and fails where $a = 2, b = 4$". Thus, less work is needed from the teacher, but the feedback for students is less abstract.

The present work will show that the test data can be automatically demonstrated with schemas $a < b$ in the first and $a \geq b$ in the latter test data due to the comparison in line 2. The feedback for a student would be nearly the same as in the manual approach. The only difference is that the demonstration for the second test is $a \geq b$ instead of $a > b$. For teachers, the difference in our approach is that manual work is not needed to demonstrate the test data. With complex programs including references, the task is far more tedious when compared to this simple example.

```
1 int distance( int a, int b ) {
2   return a - b;
3 }
```

Program 1.2: A program trying to calculate the distance between two points but failing where $a < b$.

```
1 int distance( int a, int b ) {
2   if ( a < b )
3     return b - a;
4   else if (a > b)
5     return a - b;
6   else
7     return 1; // a bug
8 }
```

Program 1.3: A program trying to calculate the distance between two points but failing where arguments are equal

We have not only reduced teachers' work with automatic demonstrations, but explained how to derive test data directly from candidate programs. Program 1.3, for example, is yet another faulty implementation for the distance problem. It returns 1 when both arguments are equal, but works correctly otherwise. Automatic assessment where test data is derived solely from the specification (manually or automatically) is unlikely to find the bug. However, the proposed approach will generate test data for each execution path of Program 1.3 and therefore will find the bug. The feedback is: "Your program passes tests where $a < b$ and $a > b$ but fails tests where $a = b$.".

Program 1.3 does not have any objects or references. The proposed solution in this thesis, however, will cover complex input structures with references (*e.g.* trees implementing dictionaries).

1.4 Scope

Automatic assessment, testing, and programming are certainly broad topics. The research question focuses on automatic demonstration of automatically created test data in programming education. At this point, we will make some justified restrictions and clarifications to our scope.

Test data without the oracle

We will concentrate on automatic test data generation only, not on whole test cases with oracles. This might sound surprising, because test data is only the first half of a complete test case. Our justification to ignore test oracles is that it is easy to create general purpose oracles for programming exercises because model solutions (*i.e.* specifications) are available. Such oracles comparing the specifications and candidate programs with given test data are often directly included into automatic assessment frameworks. Comparison can be based either the output (like in Ceilidh) or the equivalence of actual data structures (like in SchemeRobo). The reason why such oracles are not that common in industry, is that the existence of correct reference implementations is more typical for education when compared to industrial programming tasks.

General oracles can be applied even if the framework does not provide a general oracle to compare $\mathcal{P}(data)$ with $\mathcal{S}(data)$. The idea is based on automatic insertion of assertions based on post-conditions of methods. Pre and post conditions can be encoded into the specification. For some reason, this approach is typically not used in automatic assessment systems, even though benefits of checking such design-by-contract assertions are found useful elsewhere [23, 65].

Object oriented programming and Java

From several different programming paradigms and languages we have selected object-oriented programming and Java. The paradigm is found to be useful with large software because it provides various tools for encapsulation and abstraction. As proper use of such languages also forces to use these abstraction mechanisms, object oriented languages like Java often are used as the first language in CS education [44], even though typical software constructed in introductory programming courses are not so large. Because Java is already a common choice for programming education, it is reasonable to develop educational tools for such a language.

Unit-testing

By using the V-model of testing, our focus is on method level unit testing. V-model is perhaps the most widely used testing model. It separates: release, acceptance, system, interface, and unit testing from each other. In unit testing, small parts (*i.e.* units) from the evaluated program are tested independently and the co-operation among these parts is not considered. Other levels of V-mode are beyond the scope of this thesis. Interested readers are encouraged to consult any software testing book. Such books are reviewed, for example, by Grove[4].

Test data in the method level unit testing is a pair including the object state and actual parameters of the method. The object state is typically only an implicit argument named `this`. By following the standard terminology [68], units under the test (*i.e.* methods) are later on called simply programs. This not against calling specifications and candidate programs as programs in general. Program is something that can be tested and specifications can also be tested. Actually, it is a good practice to verify that specifications used in the automatic assessment are actually doing what the assignment asks to do.

Test schema and feedback

Although we discuss automatic assessment and feedback, the core of this research is on test data generation and demonstration. We will introduce a new technology we hope to be useful for programming education. Some criteria in Section 2.5.1 are related to the use in education. However, evaluating the actual educational impact is left for the future. The educational impact of this work is not the most interesting aspect because manual test data generation is already the dominant assessment approach in programming education. This work is about making test data generation easier for a teacher and to provide better test adequacy for students. Thus, automatic assessment and feedback are used to define the context in which the results of this work are applied.

1.5 Structure of this thesis

The rest of this thesis is organized as follows:

Chapter 2 is an overview about the related research: unit testing, automatic test data generation, and how symbolic execution combined with model checking

[4] `http://www.grove.co.uk/Publications/Newcomers.html` [visited April 24, 2006]

has been used for test data generation in JPF. At the end of Chapter 2, we will revise the problem statement of Section 1.1 in the light of the current test data generation approaches. We will also define the criteria to discuss the quality of the test data generation in the context of automatic assessment. discussed.

Chapter 3 introduces the main contributions of the thesis. It explains how the unit test framework in Java Pathfinder can be modified to create tests from students' programs, instead of separate specifications, and shows how the essence of a test can be demonstrated automatically.

Chapter 4 discusses the quality of the proposed solution. Pros and cons of proposed changes are considered and different JPF based test data generation techniques are compared.

Chapter 5 concludes and points out where the future research is needed.

Chapter 2

Automatic test data generation

In this chapter, we introduce the integral parts of the previous research. We formally describe some basic concepts and notations that will be used later. The nature of this chapter is more technical, compared to the introduction, which also gave some definitions, related to the motivation and scope of this work.

First, we will discuss test data generation in general. The problem can be approached from different perspectives:

1. How the selected test adequacy level can be reached (*i.e.* algorithms for that);
2. How to make test data generation computationally efficient;
3. How to create minimal test sets for the selected criteria.

A typical approach to classify techniques addressing these questions is to divide them between *black-box* and *white-box* techniques. In white-box approaches, the source code of the tested program is available whereas in black-box sources are not available. Another typical classification is to separate *heuristical* and *exact* techniques from each other. In general, heuristic approaches might include randomness and are typically more efficient. The drawback is that sometimes the result is not optimal or even missed. Exact techniques, on the other hand, will find the optimal result, but are typically slightly more inefficient. Yet another categorization is to separate specification based testing from tests created based on implementation. The last categorization was already mentioned in the introduction.

There are several ways to approach the problem of test input generation. We are not introducing them all, but concentrate on control flow oriented test generation

by using symbolic execution and a model checker. The rest of this chapter is divided as follows. Section 2.1 provides an introduction to different test adequacy metrics, in Section 2.2 we will look at different control flow oriented test data generation techniques, Section 2.3 tells about software model checking (*i.e.* a tool often used in systematic test input generation), and Section 2.4 gives an overview of the software model checker Java PathFinder that has been used for test data generation in this work. Section 2.5 will revise the problem statement and set the criteria to discuss the quality of the work.

2.1 Test adequacy criteria

To quote Brucker and Wolff [14, page 17]: "the discussion over *test adequacy criteria* [68], *i.e.*, criteria answering to the question 'when did we test enough to meet a given test hypothesis', led to more systematic approaches for *partitioning* the space of possible test data and the choice of representatives." Before introducing any of these new systematic approaches, we will describe some adequacy criteria from the software unit test adequacy survey of Zhu *et.al.* [68] and Edvardsson [22].

There are several possibilities used to define a test data adequacy criterion. We are following the approach, where a criterion is a stopping rule. It can say either that tests were adequate enough or not. Let P be a set of all programs, S a set of specifications, D a set of inputs for programs in P, and $T = 2^D$ the set of all test sets. Formally, a test adequacy criteria is a function mapping a program, specification and a test-set to a boolean value, *i.e.* C, $C : P \times S \times T \rightarrow \{0, 1\}$ [68, page 368]. Interpretations for the result are that 0=false indicates more testing is required and 1=true equals that the tests were adequate enough. In our work, specifications (teacher's reference implementation) and programs (student's solution) are both Java programs.

According to Zhu *et.al.*, there are three different approaches for constructing a test adequacy criterion:

Structural testing measures how well the different parts of the program's or specification's structure are covered during tests.

Fault-based testing criteria is based on some predefined set of faulty programs and measures how many of those faulty implementations are detected.

Error based testing stresses error-prone parts of the programs. Identifying, which

errors are more likely than the others, requires special knowledge from a tester.

Automatic test-data generators are based on some adequacy criteria in which structural and fault-based testing are, perhaps, the most popular. On the other hand, testing in CS education is, at least partially, based on a teacher's knowledge about typical errors students do. In other words, error based testing is often used in automatic assessment of programming exercises.

In this work, we have selected the structural testing approach mainly because the concepts we are further developing (*i.e.* test data generation based on symbolic execution and model checking) also use it. Due to the selected approach, and because programs and specifications are both actually Java programs, we can simplify the definition of test adequacy criteria into a function C, $C : P \times T \rightarrow \{0, 1\}$. The reason is that in structural coverage we only want different parts of programs (specifications or candidate programs) to be covered. The coverage does not directly measure the ability to find bugs. For example in the introduction, we claimed that "good test coverage for the specification does not guarantee good test coverage for the students' programs". By using the adequacy criteria $\mathcal{C}$, this is means that $\mathcal{C}(\mathcal{S}, \mathcal{T}) = 1$ does not imply $\mathcal{C}(\mathcal{P}, \mathcal{T}) = 1$, as demonstrated in Section 1.3.

In the selected approach, each item of T is one test set. In other words, a set of vectors, where each vector $\mathbf{t}$ represents a single input (*i.e.* test data) for the corresponding program. Because we are dealing with object oriented programming, the input contains both the arguments of the program (*i.e.* method) and an implicit argument `this` that is used to access the fields of the corresponding object.

2.1.1 Data flow oriented adequacy criteria

Structural testing of imperative programs is typically based either on data or control-flows, although the combination of these two is also discussed [50]. In data-flow oriented criteria, the evaluation is typically based on features like: are all the parts of the code tested where a value can be assigned into a variable (*all definitions criterion*), or how data is moving between variables. There are several tools and techniques stressing this approach, where [48, 30] are examples of the work published after the Zhu's survey. Data flow oriented techniques are not further discussed, but interested readers are encouraged to start from the surveys of Zhu *et.al.* or Edwardsson.

2.1.2 Control flow oriented adequacy criteria

A control flow graph (CFG) is an abstract representation of a program. CFG highlights the execution within a program. Nodes of CFG($\mathcal{P}$), excluding the *entry* node s and *exit* node e, are statements of $\mathcal{P}$. There is an edge from m to n if n can be executed right after m.

Formally, a control flow graph of a program $\mathcal{P}$, CFG($\mathcal{P}$) is a finite, edge-labeled, directed graph $G = \langle N \cup \{s, e\}, E, L\rangle$. G consists of a set of nodes N, edges $E = \{\langle n, m\rangle | n, m \in N \cup \{s, e\}\}$, and labels of edges L. Labels are pairs $\langle ed, f\rangle$, where $ed \in E$ and f is a function from the values of the variables of $\mathcal{P}$ into a boolean value. The boolean value is used to tell which branch should be followed when branching is possible. Values of variables are denoted by vector $\mathbf{v}$. In addition, a CFG G fulfills the following requirements:

- s has exactly one successor and no predecessors;
- e has no successors;
- For each node $n \in N$ and for each valuation of variables $\mathbf{v}$ there exist exactly one edge $ed = \langle n, m\rangle$ with the label $\langle ed, f\rangle$ such that $f(\mathbf{v}) =$ true. This means that if n has only one successors, the label of the related arc is alway true. If there are several successors, only one arc is true at a time.

The list of requirements can be reasoned as follows: Successor of s is the first statement of $\mathcal{P}$. Because $\mathcal{P}$ is unambiguous, there can be only one successor of s. Execution of the program has ended in e, thus e has no successors. The last bullet in the requirements list simply states that nondeterministic branching is not allowed. In addition, a path p can be collapsed into a single node if branching is not possible on p (*i.e.* if all the nodes of p, except the last one, have only one successor). It is an important requirement that the execution cannot branch inside a single node of CFG. For example, boolean expressions like $A \wedge B$ cannot be put inside a single CFG node if B is not always evaluated. In Java, the semantics of AND operator is defined so that in $A\&\&B$, B is executed only if A evaluates to false.

Figure 2.1 is a graphical representation from the CFG of the Program 2.1. Program calculates the greatest common divisor of non-negative integers. The example is taken from SableVM [25][1], an open-source Java virtual machine.

[1] `http://sourceforge.net/projects/sablevm` [visited December 20, 2005]

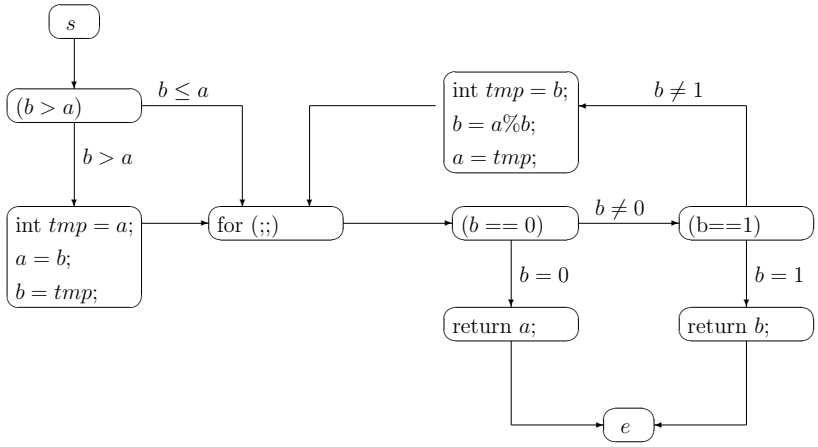

Figure 2.1: Control flowgraph of the Program 2.1. Nonmarked edge-labels are all *true*.

```
1  private static final int gcd(int a, int b)
2  {
3    // Euclid's algorithm, copied from libg++.
4    if (b > a)
5    {
6      int tmp = a;
7      a = b;
8      b = tmp;
9    }
10   for(;;)
11   {
12     if (b == 0)
13       return a;
14     else if (b == 1)
15       return b;
16     else
17     {
18       int tmp = b;
19       b = a % b;
20       a = tmp;
21     }
22   }
23 }
```

Program 2.1: A gcd method taken from BigInteger class in SableVM

Given a program $\mathcal{P}$ and an input $\mathbf{t}$ for $\mathcal{P}$, the definition of the corresponding *execution path* is: $\text{path}(\mathcal{P}, \mathbf{t}) = \langle s, a_1, a_2, \ldots a_n, e \rangle$. Nodes are in the same order as they are executed in $\mathcal{P}(\mathbf{t})$. Execution paths are also called *(absolutely) feasible paths*, opposite to *infeasible paths*.

Each input defines a feasible path, but $\text{CFG}(\mathcal{P})$ can also have several infeasible paths. Infeasibility means that there is no input that would cause such an execution. For example, the program can have dead code or some of the branches might be impossible because of the choices (*i.e.* selected branched) made previously on the path. We are using feasible path to define adequacy criteria. Thus, following definitions are not exactly the same as in [68] where unfeasible paths are included as well. To underline the difference, we are using the word *maximal*, in our definitions. For example, [68] defines the *statement coverage*, but we define the *maximal statement coverage*. Test data adequacy criteria mentioned in both surveys [22, 68] are:

Maximal statement coverage $C_{SC}(\mathcal{P}, \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\}) = 0$ iff there exists a path $\text{path}(\mathcal{P}, \mathbf{t}) = \langle s, a_1, \ldots, a_n, e \rangle$ with at least one node $a \in \{a_1, \ldots, a_n\}$ that does not belong to any of the test paths $\{\text{path}(\mathcal{P}, \mathbf{t_i}) | i \in \{i, \ldots, k\}\}$.

Maximal statements coverage holds iff all reachable statements in the code (*i.e.* nodes in CFG) are executed by at least one test.

Maximal branch coverage $C_{BC}(\mathcal{P}, \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\}) = 0$ iff there exists a feasible edge $\langle a, b \rangle$ in $\text{CFG}(\mathcal{P})$ so that $\langle a, b \rangle$ is not included in any test path $\{\text{path}(\mathcal{P}, \mathbf{t_i}) | i \in \{1, \ldots, k\}\}$.

Maximal branch coverage holds iff all the reachable edges of CFG are used by at least one test. Branch coverage is a stronger requirement than statement coverage.

Maximal path coverage $C_{PC}(\mathcal{P}, \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\}) = 0$ iff there exists an input $\mathbf{t} \notin \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\}$ so that $\text{path}(\mathcal{P}, \mathbf{t}) \notin \{\text{path}(\mathcal{P}, \mathbf{t}_i) | i \in \{1, \ldots, k\}\}$.

Maximal path coverage holds iff all the possible execution paths are included to the set of test paths (*i.e.* execution paths of the tests). Whereas branch coverage and statement coverage can always be reached, the maximal path coverage is sometimes unreachable. A program $\mathcal{P}$ having loops, can have an infinite number of execution paths, that is an infeasible requirement because test set should be finite. To solve this problem, *length-n path coverage* is defined.

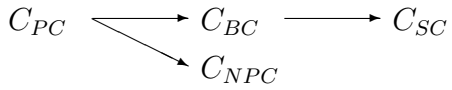

Figure 2.2: Relationships among different adequacy criteria. Notation $A \rightarrow B$ stands for $A(\mathcal{P}, \mathcal{T}) \Rightarrow B(\mathcal{P}, \mathcal{T})$.

Maximal length-n path coverage [2] $C_{NPC}(\mathcal{P}, \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_k\}) = 0$ iff there exists an input $\mathbf{t}$ so that $\text{path}(\mathcal{P}, \mathbf{t}) \notin \{\text{path}(\mathcal{P}, \mathbf{t}_i) | i \in \{i, \ldots, k\}\}$ so that $|\text{path}(\mathcal{P}, \mathbf{t})| \leq n$.

Maximal length-n path coverage holds if all the paths of length n are covered in the test set.

Maximal path coverage is the strongest requirement and maximal statement coverage is often considered the weakest. However, maximal length n-path coverage does not implicate statement coverage, or vice versa. Implications between different criteria are illustrated in Figure 2.2. For example, the fact that C_{BC}(Program 2.1, $\{[2,6],[6,2],[1,1]\}) = 1$ implies that C_{SC}(Program 2.1, $\{[2,6],[6,2],[1,1]\}) = 1$. In other words, if all the branches are executed, all the statements are also executed.

2.2 Test input generation approaches

There are several different techniques for automated test input generation. There are also numerous test generation tools introduced in the literature (*e.g.* [8, 35, 67]). Unfortunately most test input generation tools are either commercial (like JTest [61]) or unavailable for some other reason. In addition, many open source testing tools[3][4] concentrate on other aspects of testing than test input generation. Here we will not describe tools, but some techniques for test data generation. Later in Section 2.4, the techniques will be applied with JPF.

2.2.1 Method sequences *vs.* state exploration

In unit testing of Java programs, test input consists of two parts: 1) explicit arguments for the method and 2) current state of the object (*i.e.* implicit `this` pointer

[2] This criteria is actually mentioned only in [68], but because it is only a refinement to the path coverage, it is also included here.

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

[4] `http://java-source.net/open-source/testing-tools` [visited April 10, 2006]

given as an argument). The first decision in test input generation is to decide how object states are constructed (state of the current object or states of reference structures) and presented. There are at least two approaches to the task:

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

Direct state exploration tries to enumerate different (legal) input structures directly. Direct state enumeration is not the only possible approach. For example, some heuristics can be used or states can be constructed whenever needed in the program (as in Section 2.4.3) to construct different input structures.

The common justification for using method sequence exploration is that in assessment frameworks, object states can only be constructed through sequential method calls. On the other hand, in the method sequence exploration, tests are no longer testing a single method, but sequences of different methods. Testing is closer to class level unit testing than method level testing. However, the justification is not completely true. For example in Java, object hierarchies (*e.g.* states of an object) can be stored by using serialization and restored with deserialization. Both approaches are provided in standard Java libraries. However, the serialization approach has some problems addressed later in Chapter 4. The main problem is to derive tests from a specification. The corresponding specification object is serialized and the problem is how to deserialize it to an instance of class programmed by a student.

2.2.2 Symbolic execution

Another decision can be made by choosing between concrete values and symbolic values for primitives in the test input generation. The main idea behind *symbolic execution* [37] is to use symbolic values and variable substitution instead of *real execution* and real values (*e.g.* integers). In symbolic execution, return values and values of variables (*i.e.* program input and other variables) of programs are symbolic expressions consisting of symbolic input. For example, the output for a program like "`int sum(int x, int y) { return x+y; }`" with symbolic input `a` and `b` would be `a + b`.

A state in symbolic execution consist of (symbolic) values of program variables, path condition (PC) and program counter (*i.e.* information where the execution

is in the program). PC is a boolean formula over input variables and describes which conditions must be true in the state. The PC abbreviation is often used for a program counter, but we have reserved it for the path condition. A *symbolic execution tree* can be used to characterize all execution paths (*i.e.* state chains). Moreover, a finite symbolic execution tree can represent infinite number of real executions. Formally, a symbolic execution tree SYM($\mathcal{P}$) is a (possibly infinite) tree where nodes are program states during symbolic execution and arcs are possible transitions between states.

For example, the symbolic execution tree of Program 2.2, `min( X, Y )`, is illustrated in Figure 2.3. In the initial state (*i.e.* node s in the corresponding CFG), input variables have the values specified in the (symbolic) method call and PC is *true*. If the PC becomes unsatisfiable, the execution is not continued from such a node. Such nodes are marked with `backtrack` label in the figure.

All the leaf nodes of a symbolic execution tree where the PC is satisfiable represent different execution paths. Moreover, all feasible execution paths of $\mathcal{P}$ are represented in SYM($\mathcal{P}$). In the example of Figure 2.3, there are two satisfiable leafs and therefore exactly two different execution paths in Program 2.2. However, there is an infinite number of concrete executions. All satisfiable valuations for a PC (in a leaf node) will give us a real input and execution paths with all those inputs are equal.

We have claimed that a PC is a boolean formula, but can also be seen as a constraint satisfaction problem (CSP). The latter is actually a more natural interpretation because integers in programming languages are not the same as integers in mathematics. Cardinality for integers in mathematics in infinite whereas in programming languages it is fixed. For example in Java, 32 bits are used for an integer variable. Therefore, the domain of an integer variable is not unconstrained but $\{n| -2147483648 \leq n \leq 2147483647\}$. With this approach, searching satisfiable valuations is solving the CSP.

The golden age of symbolic execution goes back to 70's. The original idea was not developed for the test data generation, but formal verification and enhancement of program understanding through symbolic debugging. However, the approach had many problems [19] including:

1. Symbolic expressions quickly turn complex;
2. Handling complex data structures is difficult;
3. Loops dependent on input variables are difficult to handle.

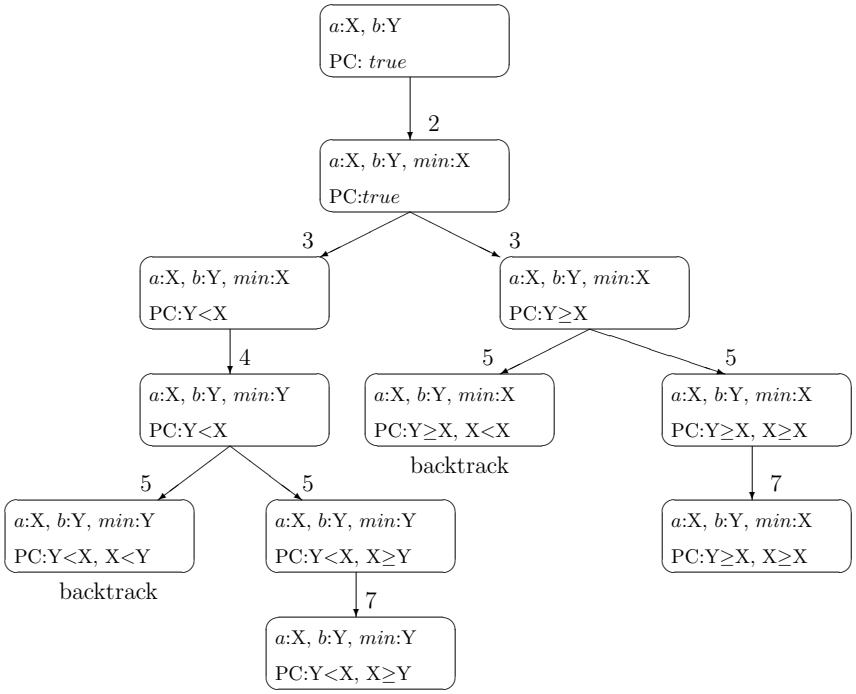

Figure 2.3: Symbolic execution tree of the Program 2.2. Numbers outside rounded boxes are referring to the line numbers of the program.

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

Program 2.2: A program calculating minimum of two arguments. Line 6 is dead code (*i.e.* newer executed) as one can see from Figure 2.3.

2.3 Model checking of programs

Model checkers are general purpose tools to research different properties from various models (*e.g.* programs). This is one of the reasons, why software model checkers are often used as a basis for test data generation software. Software model checking and model checking in general are described in this section. Practical examples how a model checker can be used in test input generation are provided in Section 2.4.

The main idea behind model checking is to prove that a desired property holds in a model under the verification or provide a counter example violating the property. A typical notation to express a property is to use a temporal logic (*e.g.* LTL or CTL). Models, on the other hand, are state transition systems or graphs where nodes are states and arcs transitions between states.

Model checking can be used, for example, in hardware verification, and software verification (distributed or concurrent). According to Visser *et.al* [64], "there has been a lot of recent interest to apply model checking with software verification [6, 7, 17, 21, 26, 28, 63]". For interested readers the article: "Model Checking Programs" [63] gives a good overview about the art of software model checking.

2.3.1 Different techniques

Model checking (*i.e.* not only software model checking) approaches are classified between *explicit-state model checking* and *symbolic model checking*. Explicit-state model checking explores all the actual states of the software individually, whereas symbolic model checking applies efficient techniques to represent logic formula (*e.g.* binary decision diagram, BDD) in order to explore several states in the same step. Explicit-state approaches are inadequate to handle complex software systems and therefore different abstraction techniques are often combined with them, to reduce the size of models. The following state space reduction techniques are typically applied on explicit-state model checking, but also with symbolic approaches.

Partial order reduction (POR) is an important technique to reduce the state space in *concurrent programs*. Instead of exploring all the different schedulings, context switches among threads are considered only when the switch can have an influence to the computation. The problem is to automate this. The technique has been developed for explicit state model checking, but applications for symbolic model checking are also researched (*e.g.* ImProvisio [42] combines symbolic model checking and POR for software model checking).

+	*pos*	0	*neg*
pos	*pos*	*pos*	{*pos*,0,*neg*}
0	*pos*	0	*neg*
neg	{*pos*,0,*neg*}	*neg*	*neg*

*	*pos*	0	*neg*
pos	*pos*	0	*neg*
0	0	0	0
neg	*neg*	0	*pos*

Figure 2.4: Semantics of + and ∗ operators in abstract interpretation if the integer domain is abstracted with negative (neg), positive (pos) and zero (0) sub-domains

State abstraction is a common term for techniques grouping states into blocks. Abstraction can add or remove behavior and therefore alter the correctness of the original model. A model is defined to be correct if there is no trace violating the verified property. It is important to note that these techniques can be applied to reduce the size of the model before model checkers are used. Therefore they can be used with either symbolic or explicit-state techniques. Abstraction techniques can be classified into:

Under approximation do not preserve correctness. If the abstracted model is correct, it does not ensure the correctness of the original model. However, an error in an under approximated model implies an error in the original model. For example, with arrays, one might want to constraint the maximum length of arrays to some real value (*e.g.* 6) instead of using symbolic length (*e.g.* N).

Over approximation adds behavior to the model. Therefore if the abstracted model is correct the original system is also correct. Correspondingly, errors in an abstracted model do not indicate errors in the original model. *Abstract interpretation* [46] is a good example of over approximation. For example, integer domain can be divided into three sub-domains where + and ∗ operators are defined as explained in Figure 2.4.

Precise approximation do not add or remove behavior (in the sense of verified property) but reduces state space. Under some restrictions, symbolic execution discussed lated is an example from precise approximation.

2.3.2 Tools

On software model checking, two methods can be identified:

1. Using a general purpose model checker and transforming the verified software into the input language of the selected model checker. Benefits of this ap-

proach are efficiency, and commonly used tools. The drawback is that the model checkers' input language might lack some constructs of the programming language (*e.g.* float variables). This would make the source to source transformation difficult.

2. Using a model checker dedicated directly for the programming language used. Such tools are perhaps less common and therefore not as efficient as some state-of-art general purpose model checkers. However, if a model checker for the programming language used is available, source to source translation is not needed and all the language constructs are directly supported.

There are several tools for software model checking and model checking in general. A comprehensive list about model checking tools in general is given, for example, in the Wikipedia[5]. We will briefly describe some state-of-art tools to give an overview about different types of tools. A Java PathFinder is, however, described with more details because it is used later in this work.

Spin [29] is perhaps the most commonly used explicit-state model checker. According to Spin's documentation[6], the input language for the system, *i.e.* PROMELA, is a "non-deterministic language, loosely based on Dijkstra's guarded command language notation and borrowing the notation for I/O operations from Hoare's CSP language". Spin has also a support for embedded C code in PROMELA models. State space reduction in Spin is mainly based on POR. Source-code (ansi C) distribution of Spin is freely available for research and educational purposes.

NuSMV [15] is perhaps the most commonly used symbolic model checker of today. NuSMV has a special input language designed to describe finite state machine (FSM) based models. FSM based approach has an implication that only finite data types (*i.e.* booleans, scalars and fixed arrays) are supported. The language is used to describe states and transitions (possibly nondeterministic ones) between states. In addition to BDDs, SAT based model checking is also supported.[7] NuSMV is open-source under LGPL license[8].

[5] `http://en.wikipedia.org/wiki/Model_checking` [Visited Jan 10, 2006]

[6] `http://spinroot.com/` [Visited February 3, 2006]

[7] Symbolic model checking techniques such as BDDs or reduced boolean circuits (RBC), both used in NuSMV, are interesting, but out of our scope. Therefore such techniques are only mentioned, but not further explained.

[8] `http://nusmv.irst.itc.it/` [Visited April 24, 2006]

Java Pathfinder (JPF) [12] is an explicit-state model checker for Java programs. The system is actually a tailored virtual machine, and therefore any compiled Java program (*i.e.* byte-code) can be directly used as an input for JPF. No source-to-source translation is needed. On the other hand, other programming languages than Java are not supported. JPF is open-source (Java) and distributed with NASA public license[9].

2.4 Test data generation with Java PathFinder

Different approaches for using JPF in test input generation are described here. As already stated, JPF is an explicit state model checker for Java programs. A big difference between JPF and other model checkers is that checked properties in JPF are not temporal logic formulas, but features described by other programs. Typical installation of JPF finds unhandled exceptions (*e.g.* assertions) and deadlocks. Checking other features can be added by implementing listeners for the state space exploration module.

In addition to standard Java libraries, JPF provides some library classes to control the model checking from the model to be checked (*i.e.* from the verified Java program). The following methods from the `Verify` class will be used later:

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

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

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

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

State matching has an important role in model checking – if several executions lead into an equal state, only one execution is continued from the state. On test input generation, state matching is used to prune similar tests.

In JPF, the equality of states is based on thread stacks and heap snapshots. This is enough, because a Java virtual machine (VM) is a stack machine. For example, model checking Program 2.1 (with a main method given by Program 2.3) for unhandled exceptions would try all the possible inputs, but actually if a state (values

[9] `http://javapathfinder.sourceforge.net/` [Visited April 24, 2006]

for variables a and b) has been seen, JPF backtracks. That is to say, input $[2, 6]$ would backtrack immediately from line `for(;;)` if the input $[6, 2]$ has already been checked before that.

```
1 public static void main(String[] args)
2 {
3   int a = Verify.random(6);
4   int b = Verify.random(6);
5   gcd(a,b);
6 }
```

Program 2.3: An example main method for the Program 2.1 to be used with JPF.

2.4.1 Explicit method sequence exploration

Explicit method sequence exploration is based on nondeterministic functions of the `Verify` class (*e.g.* program 2.1). The following phases are repeated until the chosen stopping criteria comes true.

1. The current state is stored as a possible input state.
2. A method (m) from all the methods than can change the object's state is nondeterministically chosen.
3. Arguments (arg) for the method are chosen nondeterministically.
4. Method call $m(arg)$ is executed.

For example, let us assume that we have a container implementation (*e.g.* a splay tree or a binary search tree) with methods `insert(int)` and `remove(int)`. Program 2.4 is an example of explicit method sequence exploration for constructing different states for a binary search tree. Values from 0 to 5 are inserted and removed up to 10 times.

Actually, all the possible states of a traditional binary search tree can be constructed by repeating the insert method but this is not the case with an arbitrary class. A binary search tree, where delete does not actually remove nodes, but only marks the deleted with a special "deleted" flag (*i.e.* lazy deletion is used) is such an example. For example, to get a test data with only node which is marked as deleted, calling both insert and delete is needed.

Line 8 in the program enables breaking out from the loop at any iteration, and therefore ensures that method sequences of different lengths are produced.

Actually this approach combines testing and test input generation. Tests including up to END_CRITERIA method calls are created and also executed. If states could be stored and restored (*e.g.* with serialization), different input structures constructed could be used in unit testing of a single method.

```
1  public static final int END_CRITERIA = 10;
2  public static final int MAX_ARGUMENT = 5;
3
4  public static void main(String[] args) {
5    Container c = new BinarySearchTree();
6
7    for ( int i = 0; i <= END_CRITERIA; i++ ) {
8      if ( Verify.randomBoolean() ) break;
9      if ( Verify.randomBoolean() )
10       c.delete( Verify.random(MAX_ARGUMENT) );
11     else
12       c.insert( Verify.random(MAX_ARGUMENT) );
13   }
14 }
```

Program 2.4: Test data creation with explicit method sequence exploration for a `BinarySearchTree` class with `insert` and `delete` methods.

2.4.2 Symbolic method sequence exploration

JPF provides also a symbolic execution library. The library is not yet publicly available but an evaluation version was obtained for our study. The library provides types like SymbolicInteger, SymbolicBoolean and SymbolicArray. Floats are not supported. The main idea with the library is to provide model level abstractions for programmers. For example, integer variables are replaced with SymbolicIntegers and operators between integers with methods of the SymbolicInteger class. Methods provided by the Integer class are (`_LT` ≡ `<`, `_LE` ≡ `<=`, `_EQ` ≡ `==`, `_GE` ≡ `>=`, and `_GT` ≡ `>`, `_add` ≡ `+`, `_minus` ≡ `-`, and `_mul` ≡ `*`).

Symbolic library keeps track of PC. Whenever branching depending on a symbolic variable occurs, the execution nondeterministically splits into two, and the condition (or it's negation on the else branch) is added to the PC. The framework uses a

standard CSP solver for two tasks:

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

Symbolic method sequence exploration is similar to explicit method sequence exploration. The only difference is that symbolic variables are used instead of concrete ones. Program 2.4 and 2.5 demonstrate the difference. Program 2.4 was already discussed in the previous section. Program 2.5 does the same as Program 2.4, but symbolic values are used instead of concrete ones. Because arguments given for the BinarySearchTree are no longer integers but symbolic integers, the original container class needs to be annotated before the symbolic approach can be used.

```
1  public static final int END_CRITERIA = 10;
2
3  private static void main(String[] args) {
4    Container c = new BinarySearchTree();
5
6    for ( int i = 0; i <= END_CRITERIA; i++ ) {
7      if ( Verify.randomBoolean() ) break;
8      if ( Verify.randomBoolean() )
9        c.delete( new SymbolicInteger() );
10     else
11       c.insert( new SymbolicInteger() );
12   }
13 }
```

Program 2.5: Test data creation with symbolic method sequence exploration for the annotated `BinarySearchTree` class with `insert` and `delete` methods.

2.4.3 Generalized symbolic execution with lazy initialization

Generalized symbolic execution with lazy initialization, described by Visser *et.al* [64, 36] is a symbolic state exploration technique. In contrast to method sequence exploration, the approach does not require a priori bounds for the input structures

(*e.g.* `END_CRITERIA` in Programs 2.4 and 2.5). The approach can be classified as specification based white-box testing. Even though tests are created based on Java programs, annotation is required. The annotation process cannot be completely automated and therefore it is difficult to derive test data directly from candidate programs. In Chapter 3 we will develop this technique further on to answer the challenge.

The heart of this approach is to annotate the program to be tested so that it performs the lazy initialization algorithm. The annotated program is then called with an empty `this` object as argument. The empty object means an object with uninitialized fields. In the following, we will assume that `this` is the only reference argument, but other reference arguments would be handled similarly. Uninitialized fields are initialized when they are first used (therefore the name *lazy initialization*) and the result is a test data for the original program. Actually the annotations will cause some branching (just like Verify.random in the method sequence exploration) and the result is a test set.

In lazy initialization algorithm, special getters and setters are wrote for each field of the class. After the annotation, fields are used only through these methods (*i.e.* getters and setters). Furthermore, an additional boolean field is created for each field of the original class. The boolean value tells if the corresponding field has been initialized. If the field is initialized, lazy initialization is not needed when the getter is called. Getters and setters set the corresponding boolean true. When a getter is called for an uninitialized field, lazy initialization results and the field is nondeterministically initialized to any of the following:

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

Method `_new_Node` in Program 2.6 (starting from line 7) is an example from such nondeterministic initialization. The method is called from the corresponding getter (*i.e.* `_get_next` starting from line 14). In `_new_Node`, the vector `v` contains the null object and all the objects created so far. The nondeterministic branching to select any item from `v`, or a completely new object, is on line 8.

The difference between primitive and reference fields is that each primitive is initialized to a new symbolic variable. Nondeterministic branching is not needed in the lazy initialization of primitives.

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

Program 2.6: Interesting parts from an annotated example program. The idea for the example is taken from [36] and the syntax from [64].

Lazy initialization can lead into illegal structures and therefore a conservative class invariant is required. The invariant is a method that will determine if a (partially) complete object graph can be completed into a legal one. Actually such a precondition for each method separately would be sufficient. However, if a class invariant can be defined, the same condition can be used with all the methods of the class. Terms *class invariant* and *method precondition* are therefore both used with the same meaning. Execution will backtrack if the invariant does not hold after the lazy initialization (see line 18 in Program 2.6).

The lazy initialization is best understood through an example. Program 2.7 presents a linked list, where the only method swaps two first nodes of the list if data in the first node is greater than the data in the second node. Program 2.6 is the annotated version of Program 2.7 and performs the generalized symbolic execution with lazy initialization. The idea for the example is quoted from Khursid *et.al* [36] and the annotation style and format are quoted from Visser *et.al* [64]. The `precondition` method, which is not shown, is the class invariant that would return false if there is a loop in the list.

```
1  public class Node {
2    int elem;
3    Node next;
4
5    Node swap() {
6      if ( next != null && elem > next.elem ) {
7        Node temp = next;
8        next = temp.next
9        temp.next = this
10       return temp;
11     }
12     return this;
13   }
14 }
```

Program 2.7: Linked list and a method to swap the first two nodes if the value in the current node is greater than the value in the second node

Program 2.8 is the main method to launch the test data exploration. JPF will hit the line 5 several times because of the nondeterministic branching of the annotated Node class. A partial symbolic execution tree for the swap program is provided in Figure 2.5. Only some interesting branches from the tree are taken into the figure.

A question mark "?" inside a box is for an uninitialized value (*i.e.* elem field), but otherwise stands for an uninitialized reference (*i.e.* next field). In the initial state, the object for which the swap is called is created, but the fields (*i.e.* elem and next) are uninitialized. The figure demonstrates how new objects (*i.e.* list nodes and data objects in nodes) are created by the lazy initialization as the execution goes on. The first lazy initialization is resulted from line 31 (Program 2.6). Evaluating "`_get_next() != null`" will result the lazy initialization of the next field. The next will be initialized to 1) `null`, 2) new uninitialized object or 3) pointing back to the `this` object, as illustrated by the first two rows of the figure. It the case where `next` points back to this, precondition will return false and the execution is backtracked (marked as ignore in the figure). On line 3 of the figure, the elem field of the first object gets the symbolic value $E0$. This results from calling `_get_elem` from line 31. On the same line `_get_elem` is called also for the object returned by the `_get_next`. `_get_next` has already been called and therefore lazy initialization for the next field does not happen, but the `elem` field for the second object is created (see line 4 in the figure, where E1 is introduced). The next branching results from comparing the symbolic values of E0 and E1 on line 31.

```
1 public class SwapTester {
2   public static void main(String[] args) {
3     Node n = new Node(); // at this state, n is an empty node
4     n = n.swap(); // now n is lazily initialized and modified
5     storeTestData(n);
6   }
7 }
```

Program 2.8: Main program to start test data generation for the swap method in Program 2.6. When used with Program 2.7 nothing happens because there is only one node (*i.e.* no two nodes to swap) and no lazy initialization create new nodes

Lazy initialization with symbolic values (*i.e.* generalized symbolic execution with lazy initialization) generates the symbolic execution tree of the program ($\mathcal{P}$). If SYM($\mathcal{P}$) is finite (*i.e.* the maximal path coverage is reachable) the approach will find all the leaf nodes of SYM($\mathcal{P}$), and therefore generates a test set $\mathcal{T}$ so that $C_{PC}(\mathcal{P}, \mathcal{T}) = 1$.

By using the described technique, complex object hierarchies are automatically built when unit tests for a single method $\mathcal{P}$ are generated. However, if SYM($\mathcal{P}$) is infinite, the test data generation process does not terminate. A typical case where symbolic

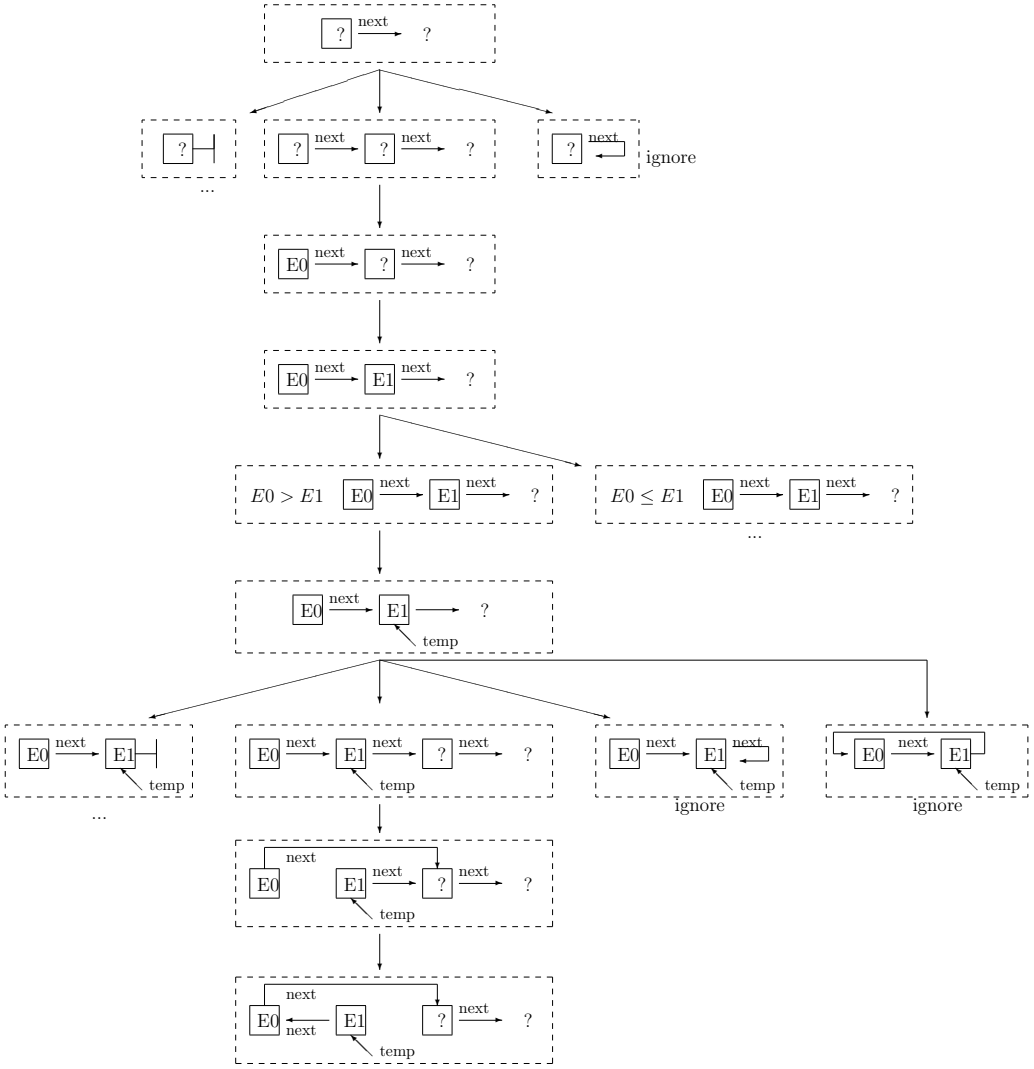

Figure 2.5: Excerpts from the symbolic execution tree of the Program 2.6. The figure is quoted (slightly simplified) from Khursid *et.al* [36, page 6].

execution tree is infinite is when the length of execution paths in the CFG($\mathcal{P}$) raises with the size of test data. One possible solution is to modify the JPF virtual machine so that only paths up to given length are checked. Another possibility is to set an upper limit for structure sizes in the class invariant.

2.4.4 Destructive updates and reconstruction in lazy initialization

Whenever a value is assigned to a variable that is already initialized, the old value is lost. We call such updates destroying an old value as *destructive updates*.

At the end of symbolic execution with lazy initialization, data structures are whatever they can be at the end of the program. However, in test data creation, we are interested about input structures, not output structures. If the program does not alter the input, input and output are the same same and we have no problem However, if the program modifies the input, we have to restore the references originally created by the lazy initialization. For example, the undermost symbolic state in Figure 2.5 is actually the symbolic end state for the input state two steps before. Both structures, the symbolic input and the structure after modifications, are in Figure 2.6. At the end of symbolic execution with lazy initialization, we have the symbolic output. The problem is how to restore the symbolic input from the symbolic output.

To overcome the problem, a mapping between original objects (*i.e.* fields created by the lazy initialization) and destructively updated objects (*i.e.* values assigned by `_set`) is needed. Original references should also be used to check the invariant. Current references cannot be used in the invariant, because the invariant is not guaranteed to be true in the middle of execution. Using current references in the invariant would therefore lead into pruning legal input structures.

Figure 2.6: Excerpts from figure 2.5. The input structure that should be reconstructed on the left and the modified structure on the right.

2.4.5 Instantiating symbolic structures

Efficient techniques for constructing concrete input structures from partially initialized (symbolic) test data are not discussed here. Symbolic expressions are not the problem because a CSP solver can be used to instantiate concrete data from symbolic variables. The problem is what should we do with unknown references (*i.e.* references that are not set by the lazy initialization). However, a naive approach for task is presented. A side product of the approach is a general purpose invariant implementation. The approach is extremely inefficient, and perhaps this is the reason why we have not seen this in the literature. With relatively small test data, typically used with programming exercises, we believe that the approach is interesting.

We will generate all (symbolic) input structures (up to given length) with symbolic method sequence exploration. Let us assume that the partially initialized object graph is subsumed into any object graph obtained through method exploration. This requires that CSPs related on both structures are not contradictory and the partially initialized graph is subgraph of the complete graph. The common root for both graphs is known (*i.e.* `this`). Test data is the corresponding complete object graph subsuming the partially initialized structure. The general purpose invariant will try to instantiate the structure and if this succeeds, the invariant holds. However, if this approach is used, we will get all the limitations of the method sequence exploration discussed in Chapter 4.

2.5 Application: Improving the automatic assessment process

The process of automatic assessment was described on a general level in Chapter 1. Now we will look at the process more deeply and point out the problems of current test data generation approaches of JPF: symbolic method sequence exploration, concrete method sequence exploration and generalized symbolic execution with lazy initialization.

A programming exercise starts from an exercise description. Based on the description, a teacher implements the specification and possibly some programs that are included to the exercise description (*i.e.* provided for students). A student implements the candidate program $\mathcal{P}$, submits it to the automatic assessment and gets the feedback that either $\mathcal{P}(x)$ failed or passed. x has traditionally been the test

data, but we have proposed that it could also be a test schema from where the actual test data is derived. To make any difference between test data and schema, it should be possible to derive several test data from a schema.

We have also stated that, in addition to traditional tests derived from the specification, test data should also be derived from the candidate program (*i.e.* $\mathcal{P}$). The program will be executed against two test sets: a test set derived from the specification and a set derived from the submitted solution. During test data generation, object states (*i.e.* test data) is serialized. When the tests are executed, the test data is deserialized to an instance of the class implemented by the student.

The problem of previous techniques is that either they do not provide abstract schemas (*i.e.* concrete method sequence exploration) or they need annotation of the code (*i.e.* symbolic method sequence exploration and generalized symbolic execution with lazy initialization). Manual annotations is problematic because tests are performed on-the-fly, *i.e.* without the influence of the teacher. Deserialization is yet another problem. How to serialize the specification so that it can be deserialized to the candidate program. Furhermore, special problems of generalized symbolic execution with lazy initialization are considered in Sections 2.4.4 (the reconstruction problem, which will be solved in Section 3.4) and 2.4.5 (the problem of instantiating symbolic structures).

In addition, special attention should be paid to tell students which tests are derived from the candidate program and which are derived from the specification. When tests are derived directly from the candidate program, a schema explicitly defines a single path in the student's program. This can (and should) be explained for students. However, when tests are derived from the specification, a schema defines a path in the specification. Thus, it is possible that there are several paths in the candidate program that can be derived from the same schema. The difference should be clear for students. However, test data derived from the specification is what hand coded test data and demonstrations have always been. Therefore, we believe that the danger of confusing students is not significant. However, if one founds tests derived from specifications confusing, it is still possible to use hand written test descriptions (*i.e.* demonstration).

Inspired by these problems of current test data generation approaches, we have set the criteria to evaluate different solutions.

2.5.1 Criteria

In addition to proposing a new automation technique for something that is traditionally done manually, the quality of the solution needs to be discussed. The different approaches (three previous from the literature and three new introduced by this thesis) of using JPF in test data generation will be compared with the criteria set here.

Preparative work

Preparative work includes tasks that the are needed before test data can be automatically derived from a Java program. With JPF, such tasks are different annotations. The discussion will compare the amount and style of annotations needed.

Generality

Automatic test data generation cannot be applied on all programs. The generality is used to discuss about such limitations. An example of possible limitations in JPF is that the symbolic execution does not support floats. This and other structural requirements for the programs to be tested are discussed. The amount of suitable cases where the test data generation approaches are applicable will be estimated.

Integrability

We would like to develop techniques and tools that are not restricted to any special assessment system, but independent tools to be used with current assessment tools (*e.g.* Ceilidh). Integrability is about how the test data generation can be integrated in automatic assessment systems. The discussion on integrability will concentrate on storing and restoring test data for automatic assessment. The topic has been selected because all the proposed data generation techniques have some problems with it.

Test case adequacy

Ala-Mutka [2] summarizes previous foundings of Kay *et.al.* [34] "it is not possible to consistently and thoroughly grade students' programs without automated assistance [...] even small programs typically have a large number of execution paths. Automation provides means to systematically cover a large number of different execution possibilities." Therefore, the ability of current test case generation techniques to

provide inputs for different execution paths should not be debilitated by the possible improvements of documentation. This criteria is measured by using different path coverage metrics introduced in Section 2.1.

Abstract feedback

According to Mitrovic and Ohlsson [45], too exact feedback can passivate learners and therefore abstract feedback should be preferred. Correspondingly, on introductory programming courses at the Helsinki University of Technology, we have observed that exact feedback (*i.e.* "program fails where $a = 2, b = 4$") guides some students to fix the counter example only. After "fixing the problem", the candidate program might work with $a = 2$ and $b = 4$, but not with other values $a < b$. Abstract feedback is described in Section 1.2: a test schema and an oracle results for $\mathcal{S}(data) = \mathcal{P}(data)$ with some data derived from the schema. Abstractness is evaluated, with the number of test data derivable from a single schema. The more test data is derivable from a single schema, the more abstract a feedback using the schema is. We will also discuss if feedback can be considered too abstract.

Chapter 3

Improvements to JPF test data generation

In this chapter we will explain our idea of using a (symbolic) path condition (PC) and (symbolic) object graphs as test schemas. In Chapter 1, we already described how such schemas are included to the feedback of automatic assessment. Roughly, the idea is that schemas demonstrate automatically created test data and therefore help students to understand tests. Schema is a generalization of test data with the property that all instantiations of a schema will lead into identical execution paths. Thus, instead of pointing a single test data leading to erroneous behavior we will demonstrate the erroneous path on a higher abstraction level. Based on the previous research of Mitrovic and Ohlsson [45], we believe that this is good for learning.

We will also introduce new approaches to turn JPF from specification based test data generation towards automatic test data generation from candidate programs. To understand how the current JPF test data generation is changed, we strongly recommend reading Section 2.4 before proceeding further on.

The content of this chapter is divided as follows: Section 3.1 is about how a PC should be simplified. Next, we will introduce two new approaches to reduce the amount of annotations in JPF test data generation: *comparable symbolic interface* in Section 3.2 and *symbolic probes* in Section 3.3. In Section 3.4, we will give a detailed solution to the reconstruction problem introduced in Section 2.4.4. Finally, Section 3.5 introduces the prototype in where some of the techniques are implemented.

Techniques addressing the annotation problem (*i.e.* Sections 3.2 and 3.3) are di-

vided so that they are actually solving slightly different problems. The comparable symbolic interface will remove simple annotations needed, for example, in symbolic method sequence exploration. Symbolic probes, on the other hand, provide interfaces for data structures in which the precondition is already defined. It follows that probes can hide the manual work of defining invariants in generalized symbolic execution with lazy initialization.

3.1 Redundant constraints

Whenever branching dependent on a symbolic variable is syntactically possible, a new constraint derived from the branching condition (or the negation of it) is added to the constraint pool. After that, JPF uses a standard constraint solving library to conclude if the PC is satisfiable and backtracks if it is not.

Sometimes previously added constraints are actually stronger than a new constraint that should be added to the PC. When tracing an execution, a human and JPF handle constraints differently. For example, a human would not be interested about the knowledge $x \leq y$ if he already knows that $x < y$. However, JPF stores all the constraints without further analysis.

Constraints that do not bring in any new information are redundant. Although a properly used repetition can be effective, we believe that the redundancy should be controlled. Our first attempt to solve the problem is not to add a constraint ϕ to the CSP Φ if existing constraints can be used to derive ϕ, in other words if $\Phi \Rightarrow \phi$.

Program 3.1 is an example of semiautomatic annotation, described by Visser *et.al* [64]. The original program is an iterative quicksort routine. With an array of 4 items, the program has 31 different execution paths and the CSP of the path 28[1] is:

$$\{\ a[2] \geq a[1],\ a[1] < a[2],\ a[0] \geq a[2],\ a[2] < a[3],\ 2 < 4,\ 2 \geq 0,\ a[1] < a[3],\ 1 < 4,\ 1 \geq 0,\ a[0] < a[3],\ 0 < 4,\ 0 \geq 0,\ 3 < 4,\ 3 \geq 0,\ a.length > 0\ \}$$

Constraints in the previous list are in reverse order. The first constraint (*i.e.* $a[2] \leq a[1]$) is the one that is added last to the PC. By using the proposed simplification scheme, the PC would be:

$$\{\ a[1] < a[2],\ a[0] \geq a[2],\ a[2] < a[3],\ a[1] < a[3],\ a[0] < a[3],\ a.length > 0\ \}$$

[1]Paths are constructed with the bfs search to the state space and numbered from 1 to 31. Path 28 is selected because it has properties suitable to demonstrate some limitations of our first proposition.

```
1  private static void quickSort(ArrayIntStructure a, Expression left,
       Expression right) {
2    Expression original_left = left;
3    Expression original_right = right;
4    Expression pivot = a._get(right);
5    do {
6      while (a._get(left)._LT(pivot)) {
7        left = left._plus(new IntegerConstant(1));
8      }
9      while (pivot._LT(a._get(right))) {
10       right = right._minus(new IntegerConstant(1));
11     }
12     if (left._LE(right)) {
13       Expression tmp = a._get(left);
14       a._set(left,a._get(right));
15       a._set(right,tmp);
16       left = left._plus(new IntegerConstant(1));
17       right = right._minus(new IntegerConstant(1));
18     }
19   } while (left._LE(right));
20   if (original_left._LT(right))
21     quickSort(a, original_left, right);
22   if (left._LT(original_right))
23     quickSort(a, left, original_right);
24 }
```

Program 3.1: Recursive quicksort after automatic annotation. All integer variables all replaced with Expressions and integer literals with IntegerConstants.

Tautologies and the constraint '$a[2] \geq a[1]$' are removed. '$a[2] \geq a[1]$' because there is a stronger '$a[1] < a[2]$' constraint just before it. However, if the constraints would have been introduced in a different order ('$\geq$' before the corresponding '$<$'), only tautologies would have been omitted. In addition, regardless of the order in where constraints are added, redundancy still exists in the simplified description. $a[3] > a[2] \wedge a[2] > a[1] \Rightarrow a[3] > a[1]$, but all the three CSPs are in the PC. In addition, $a[0] < a[3] \wedge a[0] \geq a[2] \Rightarrow a[2] > a[3]$, but again all the constraints are in the PC. A better description for the path would be for example:

$$\{\ a[1] < a[2],\ a[0] \geq a[2],\ a[0] < a[3]\ \}$$

One possible approach to remove redundant constraints better, is to check all the

constraints of the PC whenever a new constraint is introduced. First, the redundancy of a newly added constraint is checked as already described. After that, we will check if any of the previous constraints has become redundant because of the newly added constraint. This is implemented by removing all the other constraints one by one and adding them back to the CSP with the redundancy check.

Library support is needed for checking the validity of implications. For example, Omega [52] and CVC Lite [9] libraries can be used for the task as proposed by Xie *et.al* in their symbolic test input generation framework called Symstra [67]. In Symstra, the validity of implications is used to speed up the search. We have not yet implemented a direct support of the simplification in JPF, but the algorithm is given in Figure 3.1.

if $pc \Rightarrow c$ **then**
  c is redundant so do nothing
else
  $pc \leftarrow pc \cup \{c\}$
  for all Constraint $tmp \in pc - \{c\}$ **do**
    $pc \leftarrow pc - \{tmp\}$
    if $pc \Rightarrow tmp$ **then**
      tmp is redundant so do not put it back to pc
    else
      $pc \leftarrow pc \cup \{tmp\}$
    end if
  end for
end if

Figure 3.1: Algorithm for adding a new constraint (c) into the path condition CSP (pc)

By using the previous algorithms, the description for our example path, discussed throughout the section, would be

$$\{\ a[1] < a[2],\ a[0] \geq a[2],\ a[0] < a[3],\ a.length > 0\ \}$$

3.2 Comparable interface instead of annotations

Here we will describe an alternative approach for the symbolic execution where annotation is not required. The approach is based on the `java.lang.Comparable` interface. The goal is to introduce symbolic execution so that annotation or other knowledge about symbolic execution are not required during the programming.

If tests are created based on model solutions, one might argue that teachers are capable for constructing such specifications (*e.g.* Program 3.1), at least if there are semiautomatic tools supporting the process. The interface based approach, however, will especially ease the test data generation based on candidate programs instead of specifications. The motivation for this is twofold:

1. Tests based solely on model solutions might lead into scenarios where some reachable statements or branches in a student's solution are not tested.
2. By deriving tests from candidate programs, we can provide some additional feedback from the solution. For example, the quicksort example used in the previous sections might reveal that a typical worst case input for the students solution is the one in which the items are already in ascending order.

The idea is to use the `Comparable` interface instead of primitive types. `Comparable` is a standard Java library used with objects having a total ordering. The only method in the interface is `compareTo` with the following definition:

- $x < y \Leftrightarrow$ `x.compareTo(y) == -1`
- $x = y \Leftrightarrow$ `x.compareTo(y) == 0`
- $x > y \Leftrightarrow$ `x.compareTo(y) == 1`

Program 3.2 is nearly the same as Program 3.1. The differences are that it is sorting comparable objects instead of integers and that it is not annotated. A student might have implemented the program. Symbolic execution can be introduced to the program by using symbolic expressions implementing the `Comparable` interface. Students do not need symbolic comparables because they can use the program, for example, with `Integer` wrappers.

As symbolic integers of JPF already provide methods for comparison (comparison method will also store the new constraint into PC and backtrack if the PC is not satisfiable), we only need an adapter for the interface. A simple implementation for such an adapter is given in Program 3.3.

Let us consider what happens during sorting an array of length two. Simplifications will not be used. Compare method is first encountered at line 6 (Program 3.2). The execution goes into ComparableSymbolicInt and the the first (nondeterministic) branch is on line 4 (Program 3.3). In the case where -1 is returned, $a[0] < a[1]$ will

```
1  private static void quickSort(Comparable[] a, int left, int right) {
2    int original_left = left;
3    int original_right = right;
4    Comparable pivot = a[right];
5    do {
6      while (a[left].compareTo(pivot) < 0) {
7        left++;
8      }
9      while (pivot.compareTo(a[right]) < 0) {
10       right--;
11     }
12     if (left <= right) {
13       Comparable tmp = a[left];
14       a[left] = a[right];
15       a[right] = tmp;
16       left++;
17       right--;
18     }
19   } while (left <= right);
20   if (original_left < right)
21     quickSort(a, original_left, right);
22   if (left < original_right)
23     quickSort(a, left, original_right);
24 }
```

Program 3.2: Recursive quicksort with Comparable interface when annotation is not needed.

```
1  public class ComparableSymbolicInt extends SymbolicInteger implements
       Comparable {
2    public int compareTo(Object other) {
3      ComparableSymbolicInt o = (SymbolicInteger) other
4      if (this._LT(o)) return -1;
5      else if (this._GT(o)) return 1;
6      else return 0;
7    }
8  }
```

Program 3.3: Definition for a comparable type that can hide symbolic execution so that programmers should not need to care about that.

be added to the PC. Otherwise, $a[0] \geq a[1]$ is added to the PC and execution proceeds to the line 5 (ComparableSymbolicInt). The reason why $a[0] \geq a[1]$ is added is that the symbolic library, in branching, adds the condition or the negation of it to the PC. $a[0] \geq a[1]$ is the negation of $a[0] < a[1]$. The same happens in line 5: if 1 is returned, the PC will be { $a[0] \geq a[1]$, $a[0] > a[1]$ } and otherwise (*i.e.* if 0 is returned) the PC is { $a[0] \geq a[1]$, $a[0] \leq a[1]$ }. $a[0] \leq a[1]$ is added to the PC because it is the negation of $a[0] > a[1]$.

Thus, at the end of line 6 of Program 3.2, the execution has branched into three, although the comparison has only two possible outcomes (*i.e.* true or false). Use of symbolic integers would not introduce such extra branching. Executing the quicksort to the end does not introduce new branching, and at the end of execution, the PC has three possible values:

1. { $a[0] < a[1]$ }
2. { $a[0] > a[1]$, $a[0] \geq a[1]$ }
3. { $a[0] \leq a[1]$, $a[0] \geq a[1]$ }

An interesting observation is that a comparison like `this._EQ(o)` is not required in line 6 of the ComparableSymbolicInt. Negations of '<' and '>' constraints are already added to the PC, which implies the equivalence (*i.e.* $a \geq b \wedge a \leq b \Leftrightarrow a = b$). Moreover, equality of a and b is always modeled with two constraints. Thus, in order to make constraints easier to read, we should add a substitution rule $a \geq b \wedge a \leq b \rightarrow a = b$ to the simplification procedure.

In summary, the Comparable interface can hide the symbolic execution with the price of extra branching. Whereas the branching factor of comparison operators in the traditional symbolic execution is two, use of Comparable objects increases it to three. This leads into fast expansion of states and therefore execution paths and possible input structures. For example, in the prevevious section we mentioned that with arrays of four items, Programs 3.1 has 31 different execution paths. By using comparable objects (as in Program 3.2) there would be 75 paths. Another drawback is the limited computational possibilities of comparable objects when compared to integers. For example, comparable objects can not be added together due to the fact that the interface comprises only comparison methods.

3.3 Handling complex structures with probes

Next we will describe how the interface model can be extended to handle complex structures (*i.e.* data structures with references) in lazy initialization. Comparable objects can be used for primitives, just like they were used with arrays. The difficulty, however, is to hide the annotations of lazy initialization. Semi-automatic annotation, like proposed by *e.g.* Visser *et.al* [64], is one possibility, but requires hand coded preconditions. The approach we are proposing is based on the framework of super classes of fundamental data types (FDTs) hiding the class invariants, and lazy initialization. According to Korhonen [38]: "FDTs form the basis of abstractions that serve as 'archetypes' of data models. At least most of the data models used in computer science education, could be implemented in terms of these 'archetypes'. Examples of such FDTs are array, linked list, binary tree..." Probes do not directly hide symbolic execution, but the idea is to use Comparables for primitives like already described in Section 3.2.

Similar approach is used for example in the Matrix framework [39, 41][2]. The difference, however, is that Matrix uses the FDT super classes (*aka* probes) to hide algorithms animation functionality whereas we are hiding the complexity of lazy initialization.

"The most common FDTs are array, linked list, tree, and graph." [41] To provide a comprehensive framework we should provide symbolic execution probes for all of these structures. In this thesis, however, we will use only arrays and binary trees to prove our concept. In the previous section, we have already described how arrays of comparables can be used for symbolic execution. Therefore the focus of this section is on trees only. However, these principles are easily generalized for other structures, as well.

3.3.1 Assessment process in exercises using probes

The first thing to do when designing an exercise using probes is to select the correct FDT probe. Teacher implements the domain specific invariant to the probe and declares getter and setter methods as final. The exercise should be designed so that students do not need to modify getters and setters. This is because modifying these parts can easily break the lazy initialization based test data generation. After that, teacher implements the specification (*i.e.* model solution) by extending the class

[2] `http://www.cs.hut.fi/Reseach/Matrix/` [visited Mar 20, 2006]

where the invariant is defined.

A student gets the exercise description and the probe class with the invariant. The candidate program should be extended from the probe, and therefore source codes of the probe are not needed by the student (*i.e.* compiled class file is sufficient). When implementing a candidate program, student should follow the interface defined by the probe. Local variables can be freely used, but students should not add new fields to the class. The reason for this restriction is that when test data is derived from the specification, a stored instance from the specification class should be deserialized into the candidate program.

The rest of the assessment process is the same as already described in Section 2.5.

3.3.2 Binary trees

Some excerpts from the binary tree probe are given in Program 3.4. The biggest differences between the generalized symbolic execution and the probe approach are found from getters (*e.g.* `getLeft` and `getData` in our example). For comparison, a reader is referred back to Program 2.6 on page 29.

Information about whether the lazy initialization should be used is obtained from the Verify class. This makes it possible that the same probe is used for test input generation, as well as for normal execution. Implementations based on probes can be executed on normal virtual machines without existence of lazy initialization or symbolic execution. When JPF virtual machine is used, the lazy initialization can be turned on. Another possibility for doing the same would be to use different base classes for students and for test data generation. In the latter approach, all method signatures of both base classes should be the same.

Another difference between approaches (*i.e.* binary tree probe and generalized symbolic execution with lazy initialization) can be seen in the reuse of objects. A tree cannot have cycles and therefore tree-nodes are not reused. Either `null` or a new object is returned by the lazy initialization (*e.g.* line 7 in Program 3.4). This kind of structural hint possibly speeds up the search, but cannot be generated by automatic annotation. Finally, there are two abstract methods in the probe:

precondition is a class invariant or a method precondition as described in Section 2.4. It is left abstract because several data structures with different invariants can be derived from the same FDT (*e.g.* binary search tree, AVL tree and red black tree from the binary tree). In the plain binary tree, the

precondition is empty because cycles are already prevented as objects are not re-used.

getNewNode is a factory method needed in every inherited class. Inherited classes delegate the lazy initialization to the probe, thus the probe has to have a way to generate objects of the proper type. An example implementation of the method in the BinarySearchTree class is simple: `public TreeNode() {return new BinarySearhTree();}`.

```
1  public abstract class TreeNode {
2    TreeNode left = null;
3    Object data = null;
4
5    public TreeNode getLeft() {
6      if (Verify.getLazyInitialization() && !_left_is_initialized) {
7        _left_is_initialized=true;
8        left = Verify.randomBool() ? null : getNewNode();
9        _original_left = left;
10       Verify.ignoreIf(!precondition());
11     }
12     return left;
13   }
14   public Object getData() {
15     if (Verify.getLazyInitialization() && !_data_is_initialized) {
16       _data_is_initialized=true;
17       data = getNewData();
18       _original_data = data;
19       Verify.ignoreIf(!precondition());
20     }
21     return data;
22   }
23   public abstract boolean precondition();
24   public abstract TreeNode getNewNode();
25   protected Object getNewData() { return new ComparableSymbolicInt(); }
26 }
```

Program 3.4: Interesting excerpts from the binary tree probe

When the probe is used in programming exercises, the teacher will implement the precondition according to the assignment and the resulted class is provided for students. Students will extend the class with the invariant and implement other

method as explained in the assignment. An additional requirement for students is that they also have to implement the getNode (*e.g.* "`return new MyObject();`"). It is also possible to provide an outline of the class for students when the getNode can be given. The specification program is implemented just like candidate programs by extending the class with the invariant.

A common problem in method sequence exploration is how to serialize test data from specifications and deserialize it to instances of candidate classes. Probe can also answer this problem. Getters and setters of probe can be used to restore the test data for candidate programs. The approach does work if students do not change getters and setters. This can be ensured by declaring them to final[3].

3.3.3 Case study: test demonstrations in BST delete

We should also give attention to visualizing test data demonstrations for students. An object graph with symbolic primitives and constraints can be represented in various ways. As we have mentioned, probes are also used in software visualization (*e.g.* Matrix framework). Therefore it an interesting idea to combine probes of test data generation and probes of software visialization to get easy-read feedback.

Matrix framework, for example, enables saving step-by-step algorithm animations from an algorithmically modified data structure. If this is combined to the test data generation, a test demonstration could be two algorithm animations: one animation from the specification and one from the candidate program. This would not only demonstrate the input structure causing an error, but would also explain the difference between the candidate program and the specification.

At this point, we have not implemented the support of dynamic algorithm animations, but used the Matrix framework as a static back-end. Probes save input and output structures by using the text file format of Matrix. After that, Matrix or MatrixPro [33] can be used to visualize structures. Only the test data and the resulted data structure are visualized. Thus, the feedback combines three static pictures of data structures: 1) the test data, 2) resulted structure from the candidate program, and 3) resulted structure from the specification.

Binary search tree delete is used to illustrate the visualizations of Matrix. The candidate program is not listed here but the main idea in binary search tree delete can be summarized as follows: 1) search the first tree node (z) with the key to be

[3]Final methods cannot be modified in Java.

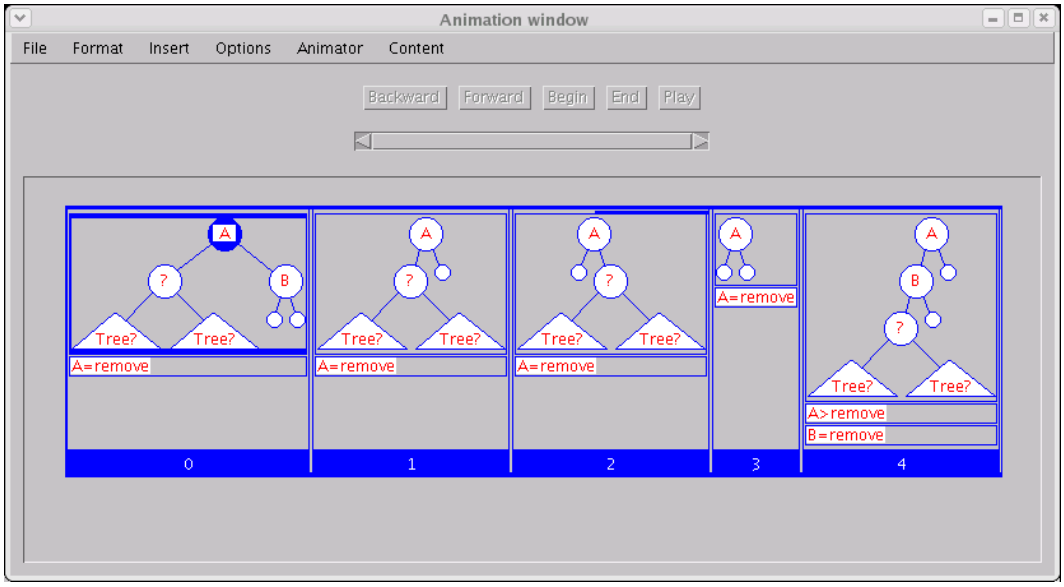

Figure 3.2: Excerpts of different input structures for the delete method of binary search trees

deleted and 2) after that, one of the following three possible scenarios is applied (*e.g.* [18]):

if z **has no children** z is replaced with null in the parent of z.

if z **has one children** z is replaced with the child of z in the parent of z.

if z **has two children** the smallest value from the right subtree (*i.e.* predecessor of z) is spliced out and the value of z is replaced with the predecessor of z. Splicing is simple, because the predecessor of z has no left children.

Figure 3.2 provides examples of the input structures. Corresponding output structures (of specification) can be seen in Figure 3.3. Figure 3.2 is a direct screen-shot from the visualization framework and Figure 3.2 is created with the latex export of Matrix. Different picture types are used to show that the output of Matrix can be combined into different automatic assessment frameworks.

Explanations of the test cases are as follows (in the same order as the cases are introduced in Figure 3.2):

- The root to be deleted has two children. The minimum from the right subtree will be spliced out and moved in place of the root.

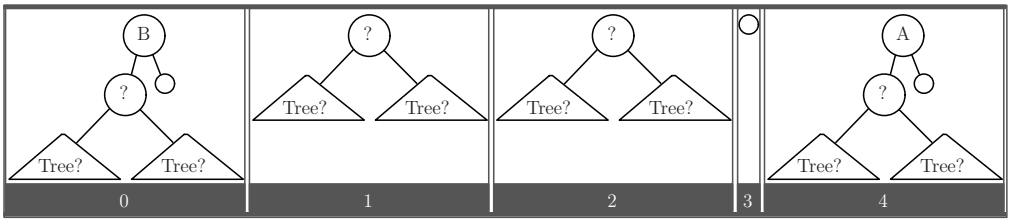

Figure 3.3: Output structures for inputs in Figure 3.2

- Two following states are nearly identical. The root to be deleted has only one child which will be the root of the new tree.
- In the fourth example, the tree has only one node. An empty tree will be returned when the node is deleted.
- In the last case, the left child of the root is deleted. Because the node to be deleted has only one child, the deleted node is simply spliced out.

3.4 Reconstruction after destructive updates

In test data generation, we are interested in input structures and therefore the values set by the lazy initialization. The reconstruction is needed because otherwise, at the end of of symbolic execution and lazy initialization, we would have symbolic output structures instead of input structures. The problem is also explained in Section 2.4.4.

The previous work [36, 64] does not explicitly define how destructive updates should be solved, but it only states that mappings between original references and destructively updated references are needed to reconstruct the original input. The requirements for the reconstruction algorithm ALSO change when test demonstration is added on test generation. If schemas are not extracted, a sufficient requirement for the reconstruction is that valid input instantiations can be derived from the symbolic structure. The additional requirement, set by the schemas, is the information about the references in the input structure that are actually used by the program. For example, Figure 3.2 provides examples of reconstructed test schemas. If schemas would not be considered, it would be sufficient to reconstruct instantiated structures such as in Figure 3.4. In the first reconstructed instantiation unknown subtrees are not empty. In other cases, unknown trees are reconstructed to empty trees.

Our reconstruction algorithm is based on storing and restoring the original refer-

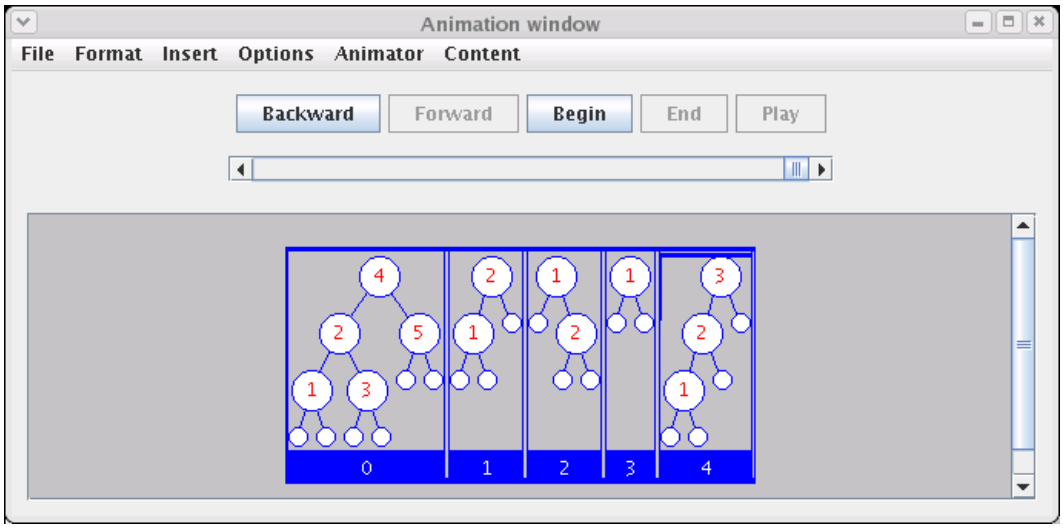

Figure 3.4: Examples of instantiated input structures from schemas in Figure 3.2

ences. For each reference field in the original structure, two additional fields of the same type, namely *original* and *current*, are used. Both fields are initialized to `null`. In addition, a variable telling the state of the reference is used. This variable has three possible values:

not initialized means that the reference has not been accessed – corresponding get or set is not called. This is the initial value for state variables.

lazily initialized means that the reference has a value, and the value was first set by the lazy initialization.

initialized by setter means that the reference has a value, but the first value of the reference was set by the algorithm – not the lazy initialization.

Getters and setters of lazy initialization with the support of reconstruction are given in Figure 3.5. The reconstruction of a reference is based on the state variable: 1) both *not initialized* and *initialized by set* imply that the original value for the field is unknown 2) if the state is *lazily initialized* the original value is pointed by the *original*. The procedure is first applied on the root object and then recursively on other objects that were successfully restored.

if *state* = not initialized **then**
  original ← `_getNew()`
  current ← *original*
  state ← lazily initialized
else
  Lazy initialization is not needed because the reference already has a value
end if
return current

if *state* = not initialized **then**
  state ← initialized by setter
else
  The reference already has a value
end if
current ← value from the argument;

Figure 3.5: Modifications for getters (left) and setters (right) of lazy initialization. `_getNew` in the getter is the nondeterministic initialization function similar to `_new_Node` in Program 2.6 on page 29.

3.5 The prototype

During this thesis project, a prototype from a framework used to ease input generation and demonstration was developed. The framework is built on the JPF software model checker, which enables backtracking and nondeterministic branching (all nondeterministic branches are executed) of execution. JPF acts not only as a model checker, but can also be understood as a metacircular interpreter: a special kind of Java virtual machine (a virtual machine with model checking capability) implemented in Java and executed inside another Java virtual machine (JVM).

The framework provides implementations for simple data structures. These data structures can be used to implement various algorithms and programs. When a program using the data structures provided by the framework is executed inside the JPF, test input structures up to given length are generated and demonstrated. Test inputs can be used when the same program (or another program with the same interface) is executed inside a standard JVM. Figure 3.6 illustrates this.

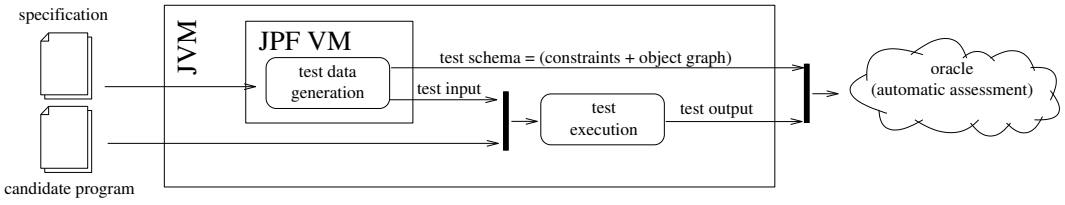

Figure 3.6: Execution inside JPF provides test input for traditional testing (execution inside a standard JVM)

In the figure, the program to be tested and the model program are drawn as separate entities. However, the same program can be used in both roles. Moreover, we believe that for adequate testing, test data should be derived from both, the program to be tested as well as a correct model solution. The framework provides two inputs for the automatic assessment system: the test data and the corresponding schema. How the assessment system evaluated the correctness of a test is not in the scope of this thesis but comparing the data structures or the outputs are typical approaches as already described in Chapter 1.

The package architecture of the framework is simple. The symbolic execution framework of JPF already provides the following packages:

gov.nasa.jpf.symbolic.integer is the biggest package in the framework. Classes for symbolic integers, operators, path conditions, and the interface to communicate with the constraint solver are found here.

gov.nasa.jpf.symbolic.array provides a class for symbolic arrays. In the symbolic array, not only the values, but also indexes are symbolic integers.

gov.nasa.jpf.symbolic.string provides a class for symbolic strings. The interface for symbolic strings is similar to the standard `String` class, but branching possibilities are recorded into the path condition.

We have made the following additions to the package hierarchy:

gov.nasa.jpf.symbolic.probe.fdt is intended for fundamental data types (*i.e.* linked list, tree, and graph) with the support of lazy initialization in JPF. The prototype provides implementation for binary trees only, but other structures would be similar.

gov.nasa.jpf.symbolic.probe.cdt is for partially complete conceptual data types (CDTs) inherited from the the FDT package. CDTs are reference implementations of abstract data types (ADT). Binary search tree, for example, is a reference implementation for the dictionary ADT. However, classes in this package are not complete CDTs. The actual functionality is missing, but the interface and the class invariant are defined. The reason why the package is named as CDT instead of ADT is due to the invariant. Although the functionality is missing, the invariant limits the implementation logic more than ADT would. The prototype provides a binary search tree implementation for

the dictionary interface. The binary search tree is extended from the binary tree of FDT package.

The model solution and students' solutions are then inherited from a class where the invariant is defined (*i.e.* gov.nasa.jpf.symbolic.probe.cdt). The only requirement for solutions is that fields are not to be accessed directly, but getters and setters are always used. The framework ensures this by using proper visibilities. In addition, getters and setter can be defined *final* in the precondition class to prevent students from re-defining those and thereby destroying lazy initialization.

One solution for the infinite execution problem raised by the size of input structures is to modify JPF search strategies (*i.e.* paths with certain number of byte code instructions will backtrack). Another possibility is to limit the size of input structures with invariant. For the prototype, we have selected the latter, because in the approach, a user of the framework is not required to have any special insight about JPF.

3.5.1 MJI interface to exchange information between tests

Execution in JPF is typically slower than executing the same code in another JVM. To solve this efficiency problem (and some other problems also) JPF provides a mechanism called a Model Java Interface (MJI) to transfer execution from the JPF virtual machine to the underlying JVM. The mechanism is analogous for Java Native Interface (JNI), which makes possible to call native (*i.e.* operating system specific) routines from Java. The drawback of MJI is that model checking behind the interface is not possible.

JPF symbolic execution framework uses MJI, for example, to communicate with the constraint solver. Efficiency, however, is not the only possible reason for using MJI. Whenever backtracking happens, information from the state from where we backtracked is lost – no matter if the reason for backtracking was pruning an execution branch or end of execution. On the other hand, construction of input for the Matrix visualization system requires exchanging information among tests (*e.g.* how many tests we have created so far). In such cases, we have used MJI. Actually graphical routines are also a typical reason for using JNI. Similarity of MJI and JNI is illustrated in Figure 3.7. The figure is adopted and modified from JPF online documentation.

It would also be possible to use some add on scripts to modify the input of Matrix.

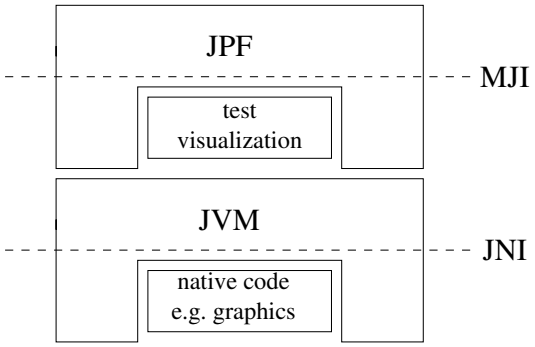

Figure 3.7: Similarity of Java Model Interface (JMI) and Java Native Interface (JNI) to provide access to the underlying virtual machine.

In that case, the MJI would not be needed. Usage of MJI is, however, justified because our ultimate goal is to integrate the testing framework more deeply into Matrix. Instead of static visualizations from test input structures, we are interested in dynamic algorithm visualizations where modifications for a symbolic input structure are shown step-by-step.

Chapter 4

Discussion

In this chapter, we will compare different ways to use JPF in test data generation with the criteria in Section 2.5.1. The title of this chapter is "Discussion", instead of "Results", because we are not presenting exact measurements. The criteria are more like an outline helping to identify characteristic features of different test data generation approaches in automatic assessment.

Three fundamentally different approaches for using JPF were described in Section 2.4: 1) explicit method sequence exploration 2) symbolic method sequence exploration, and 3) generalized symbolic execution with lazy initialization. In Chapter 3, we introduced two new approaches: 1) the Comparable interface in Section 3.2, 2) and probes in Section 3.3. The new techniques are designed to help test data generation directly from students' candidate programs. They can (and should) also be combined with previous techniques. Figure 4.1 summarizes the resulting six[1] different test data generation approaches.

On upper level, we have separated techniques between *method sequence exploration* and *lazy initialization*. In method sequence exploration, method sequences are constructed in order to get versatile input structures. Lazy initialization uses only the method under the test and visits the whole symbolic execution tree of the method, if possible. In addition, symbolic execution is used with lazy initialization but it can also be used in (symbolic) method sequence exploration.

Symbolic execution and lazy initialization both need different types of annotations. New techniques we have developed are hiding these annotations from users. Comparable interface answers to the challenge of symbolic execution and probes to the

[1] All combinations are not reasonable

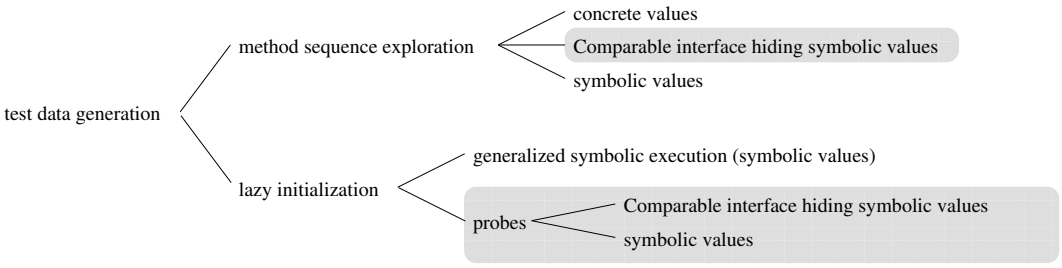

Figure 4.1: Different test data generation approaches under discussion: new techniques introduced in this work are on gray background, whereas techniques on white background are from the related (previous) research.

lazy initialization specific problems.

We have also demonstrated how constraints of symbolic execution can be simplified in order to make them easier to understand for students. Quality of the proposed constraint simplification is not discussed in the following sections. It is understood as an obligatory phase when constructing test schemas. Quality of the simplification should be evaluated by considering to how student benefit from it. This kind of educational study is mentioned in Section 5.2.1, future research. Currently, we can only state that the simplification makes schemas readable for humans, when compared to unsimplified schemas. The open question is, how much we should simplify?

4.1 Preparative work

The preparative work is evaluated based on the amount of annotations required. The scale includes values *none*, *automatic*, and *semi-automatic*. None means that absolutely no annotation is needed, automatic means that the annotation process can be automated, and semiautomatic means that the annotation process can be partially automated, but substantial handiwork is still needed. Table 4.1 summarizes our observations in this category.

Method sequence exploration requires some annotation if symbolic arguments are not hidden behind the Comparable interface. However, the annotation process can easily be automated as variables of int type are only replaced with SymbolicInteger variables and operations between integers are replaces with method calls. Visser *et.al* [64] have already described a semiautomatic tool for the task. Actually the tool can also construct additional fields needed in generalized symbolic

Table 4.1: Evaluation against the criterion of preparation work

technique		annotation needed
Method seq. exploration	with concrete	none
	with comparables	none
	with symbolic	automatic
Lazy initialization	generalized symbolic	semiautomatic
	probes with comparables	none
	probes with symbolic	automatic

execution with lazy initialization as well as getters and setters for fields. Use of fields are also replaced with getter calls and definitions (*i.e.* assignments) with calls to corresponding setters. The only task in the tool that is not automated is the type analysis. Another, more dramatic, drawback is that the annotation tool is not freely available[2]. Eclipse framework, however, provides a good framework to implement such a tool as it has a direct support for type analysis and constructing abstract syntax trees [57].

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

4.2 Generality

Generality is about what kind of programs can be used as a basis in test data generation. Table 4.2 gives relative ranking between techniques – more stars in the figure indicate that there are more situations in which the technique can be applied.

Method sequence exploration has practically no limitations and is therefore ranked to the highest place. Rest of the techniques are first ranked according to the com-

[2]The tool has been reported in [64] but we have not found that the tool to actually downloadable.

Table 4.2: Evaluation against the criterion of generality

technique		generality
Method seq. exploration	with concrete	*****
	with comparables	**
	with symbolic	****
Lazy initialization	generalized symbolic	****
	probes with comparables	*
	probes with symbolic	***

parable *vs* symbolic is the first classification. Use of symbolic objects is considered more general approach when compared to Comparables. The secondary classification criteria has been the use of probes. If probes are not needed, it is considered more general when compared to cases where the program is build on probes.

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

Probes can also limit the generality. A new probe is needed for every possible data structure, which limits the number of supported programs. It is true that there are only few FDT structures, but the problem is that a probe should also implement the invariant. This makes it difficult to use general purpose FDT probes, because the invariant is governed by the domain where the FDT is used.

Table 4.3: Evaluation against the integrability criterion

technique		annotation needed
Method seq. exploration	with concrete	easy/difficult
	with comparables	easy/difficult
	with symbolic	easy/difficult
Lazy initialization	generalized symbolic	difficult
	probes with comparables	moderate
	probes with symbolic	moderate

4.3 Integrability

This category measures how easily test data generation techniques can be integrated into existing automatic assessment tools. As the test data generation is only a front-end for automatic assessment, the evaluation is based on how easily test data can be exported for automatic assessment. The scale includes values easy, moderate, and difficult. In some approaches two values are marked, which indicates that the integration process depends on how the test data generation techniques is used. Results are summarized in Table 4.3. For method sequence techniques, two values are marked. First evaluation is for the case when the approach is used for testing only (*i.e.* class level unit testing). The second value is for method level unit testing where structures are stored and reused as test data.

Integration is easy when method sequence exploration is not used to save states. Storing method sequences to a text file is sufficient. Furthermore, test data generation can generate test programs for different inputs automatically. The problem with the approaches marked as difficult is related to restoring states. One can easily serialize and deserialize tests derived from the program to be tested. If tests are derived from a model solution the deserialization is problematic. Probes provide a common interface for the model solution and students' solutions. Getters and setters enable an easy way to store and restore states among programs derived from the same probe. Probe based approaches are therefore marked as moderate, instead of difficult.

4.4 Test adequacy

Different structural coverage metrics are defined in Section 2.1. As explained in Section 2.4.3, lazy initialization can provide length-n path coverage. On the other hand, method exploration approaches can provide all different input structures up to given size. However, the size in method sequence exploration is defined with method calls needed to construct the state. Assuming a program $\mathcal{P}$, it is possible that a small state ($\mathbf{t}$) (measured with the number of objects) is reached only through a long method invocation sequence. At the same time, $|\text{path}(\mathcal{P}, \mathbf{t})|$ can be small, or even minimal (*i.e.* there exists no test data $\mathbf{t_2}$ so that $|\text{path}(\mathcal{P}, \mathbf{t_2})| < |\text{path}(\mathcal{P}, \mathbf{t})|$). As a summary, lazy initialization provides better adequacy when compared to method exploration techniques. Theoretically all the approaches using the lazy initialization can provide the same test adequacy. The same is also true in the group of method sequence exploration techniques. Method sequence exploration based on symbolic or concrete variables can provide the same test adequacy, although there can be some differences in the efficiency.

There are efficiency differences inside the main approaches (*i.e.* lazy initialization and method sequence exploration). Using concrete values is the most inefficient approach. The state explosion is fast. Symbolic techniques are therefore superior when compared to the concrete method sequence exploration. A proper efficiency evaluation between probes and generalized symbolic execution waits to be done, but our preliminary results indicate that structural hints (*e.g.* nodes are not reused in the case of trees), used with probes, make the probe approach more efficient when compared to generalized approach.

All the adequacy discussed so far is about testing a program from where tests are derived. However, we believe that when a student's program is tested, tests should be created 1) based on a model solution that is known to be correct and 2) after that by using the program to be tested. A "correct" model is used to ensure that all input structures leading to different kind of behavior are actually generated. For example, if the student's program is oversimplified (*i.e.* some special cases are not checked), tests derived only from it would lead into situation where all the branches of the definition would not be checked. Correspondingly, if a program to be tested has a different branching strategy than the model solution, some reachable[3] statements or branches might be missed.

[3] reachable by using an input following the input size limitation used

Table 4.4: Evaluation against the criterion of abstractness

technique		abstractness
Method seq. exploration	with concrete	*
	with comparables	**
	with symbolic	**
Lazy initialization	generalized symbolic	***
	probes with comparables	***
	probes with symbolic	***

4.5 Abstract feedback

We are proposing test schema to be used when constructing feedback for students. We have also stated that abstract feedback is preferable, when compared to too exact feedback. In this category, the evaluation is based on how many test data can be derived from the same test schema. In other words, how general the schema is. All the described approaches have a property that executions leading into two different execution paths cannot be derived from the same schema. Table 4.4 gives the relative ranking between techniques – more stars in the figure indicate that the schemas are more general.

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

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

A reason why the concept of redundancy is interesting is that even with the most abstract approaches some redundancy exists. The extra branching, and therefore redundancy, that the Comparable interface brings was described in the previous

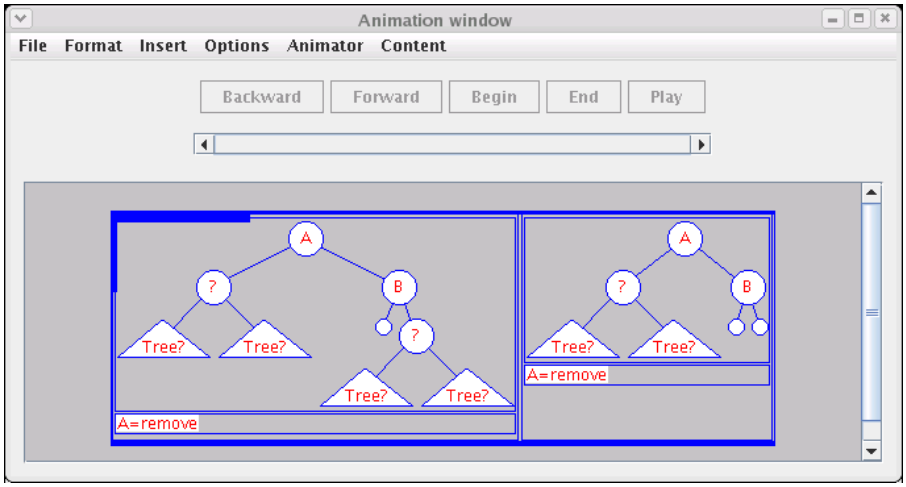

Figure 4.2: Two input data for the binary search tree leading to identical execution paths.

chapter. When comparison of symbolic integers has two possible values the comparison of Comparable objects had three possible outcomes.

```
1  if( node.getLeft() != null && node.getRight() != null ) {
2    BSTNode minParent = node;
3    BSTNode min = (BSTNode)node.getRight();
4    while (min.getLeft() != null) {
5      minParent = min;
6      min = (BSTNode)min.getLeft();
7    }
8    node.setData(min.getData());
9    if (node == minParent)
10     minParent.setRight(min.getRight());
11   else
12     minParent.setLeft(min.getRight());
13 }
```

Program 4.1: Excerpts from the binary search tree delete routine

Nondeterministic branching in lazy initialization will also add extra branching to the program. An example can be given by recalling the BST example from Section 3.3.3. Program 4.1 provides is the case when a node to be deleted has been found and it has two childs. Both input structures in Figure 4.2 are obtained through the lazy initialization with probes. The node to be deleted is A in the both cases. In both

cases, B is the smallest value in the right subtree of A. Thus, B is spliced out, by setting the link (originally pointing to B) in the father of the B to the right child of B. The right child of B is accessed. As a consequence, it is initialized to `null` or a new object. Because the `right` pointer in A is simply set to the right child of B, the execution is the same regardless of the value.

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

Chapter 5

Conclusions

5.1 Evaluating the results

Automatic test data generation and demonstration in the context of programming exercises and automatic assessment have been studied. The work presents a novel idea of extracting test schemas and test data Test schema is defined to be an abstract definition from where (several) test data can be derived. The reason for separating these two concepts is to provide automatic demonstrations from automatically produced test data and therefore from what is tested.

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

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

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

Both new approaches are also a step from model based testing towards test creation based on working Java programs. As pointed out in Sections 1 and 4, in automatic assessment tests should be created from both model solution and student's solution.

Therefore model based testing with manual annotations is not the best choice for courses with several hundreds of students. Probes and Comparable interface hiding the need of annotation are some possibilities to make such automatic test data generation techniques more attractive in education.

The discussion pointed out that method level unit testing is difficult for complex structures. Method sequence exploration techniques can be used to produce series of method invocations – in which case the test does not concentrate on a single method. The research has pointed out that probes can overtake this problem – only methods that are known to be correct are used when reconstructing the test data. In addition, test data can be derived either from the specification or the candidate program.

If the state of an object is wanted to be saved and reused as test data for a single method, use of probes is found to be the best approach.

Although we have selected Java to be the language of our study, our results can be extended to other languages and paradigms. The work concentrates on object oriented languages. As illustrated in Figure 5.1, object oriented approach is a subcategory of imperative programming. The addressed techniques are therefore usable within other imperative paradigms and even procedural languages (*e.g.* C or Fortran), but not with functional (*e.g.* Lisp or Scheme), database related (*e.g.* SQL) or formal (*e.g.* Turing machines) languages. Concurrency is not discussed in the results of this thesis. With JPF, however, it is also possible to generate test data for concurrent programs. Thus, concurrent programming is partially included to scope of this work in the figure.

Automatic assessment of programming exercises is not the only domain where the results of this work can be applied. Other possibilities are for example:

Tracing exercises is another educational domain where the presented techniques can be directly applied. In tracing exercises test data and algorithm are given for a student. The objective is to simulate (or trace) the execution (*e.g.* [40]). The problem of test adequacy (*i.e.* providing test data for students) is the same as addressed in this research.

Traditional test data generation can also benefit from our results. We believe that the idea of hiding the symbolic execution behind the Comparable interface is interesting. Extra branching resulting from the Comparable construction is not that bad, because it is nearly the same as boundary value testing (*e.g.* [27]).

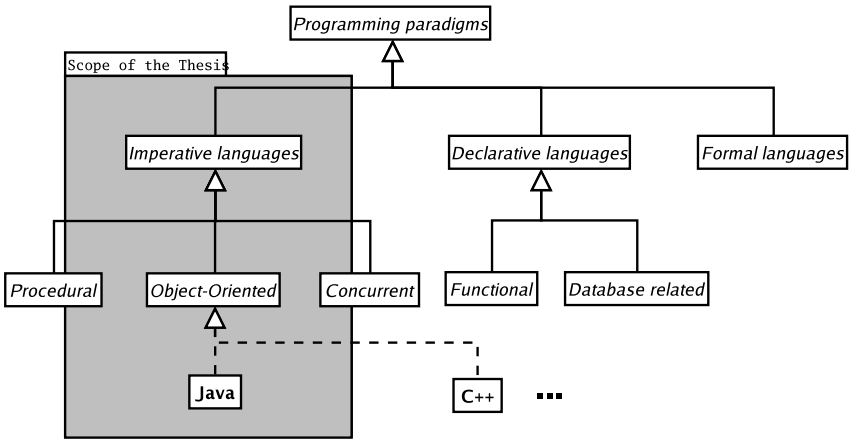

Figure 5.1: Relations between the language of the scope (*i.e.* Java) and programming paradigms. The taxonomy in the picture is taken from the book *Programming Languages* by Appleby and Vandekopple [4]. In this taxonomy, the characteristic feature for imperative languages is the concept of variables.

> Instead of creating one test data for a path with the constraint $a \leq b$, two tests are created: $a = b$ (*i.e.* the boundary value test) and $a < b$.

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

5.2 Evaluating the research

We believe that strengths of the research is identifying different possibilities of using JPF in test data generation. The related literature survey is also comprehensive.

Although the results are promising, criticism can be presented about what results are included to this study. In order to provide better technical evaluation, more comprehensive efficiency studies should be performed. In the context of education we were only interested that with small data structures the techniques are fast enough. Here, fast enough means that they can be included to the chain of automatic assessment (*i.e.* results should be obtained within seconds rather than minutes). In

order to be a better educational research, an educational research setup is needed. In Chapter 1 we ruled the educational evaluation out from the scope but we did not remove the need of such study.

Next we will give more examples about possible topics of future research.

5.2.1 Future research

Technological point of view

The proposed technology requires explicit definition (and implementation) for the class invariant. The task is not required from a teacher but from the person implementing a probe. Although the task is not typically assigned to a teacher, it is time consuming. Techniques to automate invariant construction are needed. In Section 2.4.5 we described a naive approach to use symbolic method sequence exploration to construct valid object hierarchies from partially initialized structures. We also mentioned that the same approach can be transferred for invariant construction. If a partially initialized object graph is subsumed into an object graph obtained through method exploration, the invariant holds in the partially initialized structure. We believe that the approach is worth more research as it could replace explicit invariant construction.

Completely automated invariant detection techniques are also studied (*e.g.* Perkins and Ernst [49]). Such techniques are typically used to get better understanding on the software. It is an open question if such techniques are suitable for state exploration based test data generation. A concrete example from a publicly available invariant generation system, that could be used in the future research, is the Daikon software[1].

Educational and psychological point of view

More insights about how students would use and benefit from automatic test demonstrations are definitely needed. Although we have described a technically interesting methodology to integrate automatic test input generation and documentation, the only purely objective evaluation is based on technical properties (*e.g.* efficiency) of the solution. However, we believe that our criteria gives a good basis for further studies. The next step would be to study how the selected approach affects to users.

[1] `http://pag.csail.mit.edu/daikon/` [Visited April 19, 2006]

Possible contextual frameworks for future studies are mental models [47] and models about program understanding in debugging [66].

Developing the framework

Currently the framework is not integrated into any automatic assessment system intended for programming exercises.

The symbolic execution framework of JPF is not yet public. The unfortunate consequence is that software developed for the thesis cannot be put into public domain. We are hoping to see symbolic execution framework published and after that our work could be polished into publishable format.

Currently the framework is a proof of concept with an limited support for different data structures. FDT level structures supported are arrays and binary trees. The only implemented CDT structure is binary search tree. Thus, more structures are needed.

Bibliography

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, New York, NY, USA, 2005. ACM Press.

[2] Kirsti Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15:83–102, 2005.

[3] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Järvinen. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3:245–262, 2004.

[4] Doris Appleby and Julius J. Vandekopple. *Programming Languages: paradigm and practice*. McGraw-Hill College, 2nd edition, 1997.

[5] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Păsăreanu, Grigore Rosu, and Willem Visser. Experiments with test case generation and runtime analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Proceedings of Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003*, volume 2589 of *Lecture Notes in Computer Science (LNCS)*, pages 87–108. Springer-Verlag, 2003.

[6] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[7] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 1–3, New York, NY, USA, 2002. ACM Press.

[8] Michael Barnett, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Validating use-cases with the asml test tool. In *Proceedings of 3rd International Conference on Quality Software*, pages 238–246. IEEE Computer Society, 2003.

[9] Clark W. Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker category b. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 2404 of *Lecture Notes in Computer Science (LNCS)*, pages 515–518. Springer Verlag, 2004.

[10] Mordechai Ben-Ari. The bug that destroyed a rocket. *Journal of Comptuer Science Education*, 13(2):15–16, 1999.

[11] Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and Abdullah Mohd Zin. Ceilidh: A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*, Vienna, Austria, 1993.

[12] Guillaume Brat, Willem Visser, Klaus Havelund, and SeungJoon Park. Java pathfinder - second generation of a java model checker. In *Proceedings of the Workshop on Advances in Verification, Chicago, Illinois*, July 2000.

[13] Stina Bridgeman, Michael T. Goodrich, Stephen G. Kobourov, and Roberto Tamassia. Pilot: an interactive tool for learning and grading. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 139–143, New York, NY, USA, 2000. ACM Press.

[14] Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing: 4th International Workshop*, volume 3395 of *Lecture Notes in Computer Science (LNCS)*, pages 16 – 32, Linz, Austria, January 2005. Springer-Verlag.

[15] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In

Proc. International Conference on Computer-Aided Verification (CAV 2002), volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

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

[17] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[18] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

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

[20] Ole-Johan Dahl, Edsger W. Dijkstra, and C.A.R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press, 1972.

[21] Claudio DeMartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent java programs. *Softw. Pract. Exper.*, 29(7):577–603, 1999.

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

[23] Stephen H. Edwards. Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential. *Software Testing, Verification & Reliability*, 10(4):249–262, 2000.

[24] John English. Automated assessment of gui programs using JEWL. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 137–141. ACM Press, 2004.

[25] Etienne M. Gagnon and Laurie J. Hendren. Sablevm: A research framework for the efficient execution of java bytecode. In *JVM'01: Proceedings of the Java Virtual Machine Research and Technology Symposium*. usenix, April 2001. available online from `http://www.usenix.org/events/jvm01/` [visited Dec 19, 2005].

[26] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM Press.

[27] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[28] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with blast, 2003.

[29] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[30] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 232–242, Washington, DC, USA, 2003. IEEE Computer Society.

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

[32] W. H. Jessop, J. Richard Kane, S. Roy, and J. M. Scanlon. ATLAS - an automated software testing system. In *ICSE*, pages 629–635, 1976.

[33] Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke. MatrixPro - A tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33, The University of Warwick, UK, July 2004.

[34] David G. Kay, Terry Scott, Peter Isaacson, and Kenneth A. Reek. Automated grading assistance for student programs. In *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, pages 381–382, New York, NY, USA, 1994. ACM Press.

[35] Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[36] Sarfraz Khursid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff,

editors, *Proceedings 9th International Conference on Tools and Algorithms for Construction and Analysis*, volume 2619 of *Lecture Notes in Computer Science (LNCS)*, pages 553–568. Springer-Verlag, April 2003.

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

[38] Ari Korhonen. *Visual Algorithm Simulation.* Doctoral dissertation (tech rep. no. tko-a40/03), Helsinki University of Technology, 2003.

[39] Ari Korhonen, Lauri Malmi, and Riku Saikkonen. Matrix — concept animation and algorithm simulation system. In *Proceedings of The 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, page 180, Canterbury, United Kingdom, 2001. ACM Press, New York.

[40] Ari Korhonen, Lauri Malmi, and Panu Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of Kolin Kolistelut / Koli Calling – Third Annual Baltic Conference on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

[41] Ari Korhonen, Lauri Malmi, Panu Silvasti, Ville Karavirta, Jan Lönnberg, Jussi Nikander, Kimmo Stalnacke, and Petri Ihantola. Matrix - a framework for interactive software visualization. Research Report TKO-B 154/04, Laboratory of Information Processing Science, Department of Computer Science and Engineering, Helsinki University of Technology, Finland, 2004.

[42] Flavio Lerda, Nishant Sinha, and Michael Theobald. Symbolic model checking of software. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[43] Nancy G. Leveson and Clarke S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[44] Renée McCauley and Bill Manaris. Computer science education at the start of the 21st century – a survey of accredited programs. In *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference*, pages F2G10–F2G15, 2002.

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

[46] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[47] Donald A. Norman. Some observations on mental models. In D. Gentner and Al Stevens, editors, *Mental Models*, pages 7–14. Lawrence Erlbaum Associates, 1983.

[48] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Trans. Softw. Eng. Methodol.*, 13(2):199–239, 2004.

[49] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2004.

[50] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, 1990.

[51] Corina S. Păsăreanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *Proceedings of 11th International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science (LNCS)*, pages 164–181. Springer-Verlag, 2004.

[52] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[53] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Software Eng.*, 2(4):293–300, 1976.

[54] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of The 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*, pages 133–136, Canterbury, UK, 2001. ACM Press, New York.

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

[56] Otto Seppälä, Lauri Malmi, and Ari Korhonen. Observations on student errors in algorithm simulation exercises. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 81–86. University of Joensuu, November 2005.

[57] Mariana Sharp, Jason Sawin, and Atanas Rountev. Building a whole-program type analysis in eclipse. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 6–10, New York, NY, USA, 2005. ACM Press.

[58] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' java programs. In *Proceedings of the sixth conference on Australian computing education*, pages 317–325. Australian Computer Society, Inc., 2004.

[59] Alan Mathison Turing. Correction to: On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(43):544–546, 1936.

[60] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[61] G. Venkat. Product review: Jtest 7.0 From Parasoft. *Java Developers Journal*, June 2005.

[62] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203 – 232, April 2003.

[63] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[64] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM Press, 2004.

[65] Jeffrey Voas. How assertions can increase test effectiveness. *IEEE Software*, 14(2):118–119,122, 1997.

[66] Anneliese von Mayrhauser and A. Marie Vans. Program understanding behavior during debugging of large scale software. In *ESP '97: Papers presented at the*

seventh workshop on Empirical studies of programmers, pages 157–179, New York, NY, USA, 1997. ACM Press.

[67] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005.

[68] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.