

Jan Lönnberg

Visual testing of software

7th October 2003

Teknillinen korkeakoulu Helsinki University of Technology
Tietotekniikan osasto Department of Computer Science and Engineering

Helsinki University of Technology
Abstract of master's thesis

Author:	Jan Lönnberg
Title of thesis:	Visual testing of software
Date:	7th October 2003
Number of pages:	1 + x + 88
Department:	Department of Computer Science and Engineering
Professorship:	T-106 (Software Technology)
Field of study:	Software Systems
Supervisor:	Lauri Malmi
Instructor:	Ari Korhonen
<p>Software development is prone to time-consuming and expensive errors. Finding and correcting errors in a program (<i>debugging</i>) is usually done by executing the program with different inputs and examining its intermediate and/or final results (<i>testing</i>). The tools that are currently available for debugging (<i>debuggers</i>) do not fully make use of potentially useful visualisation and interaction techniques.</p> <p>This thesis presents a new interactive graphical software testing methodology called <i>visual testing</i>. A programmer can use a visual testing tool to examine and manipulate a running program and its data structures.</p> <p>Systems with techniques applicable to visual testing in the related domains of debugging, software visualisation and algorithm animation are surveyed. Techniques that are potentially useful to visual testing are described, examined and evaluated, and a design for a visual testing tool based on these techniques is presented. The tool combines aspects of user-controlled algorithm simulation, high-level data visualisation and visual debugging, and allows easier testing, debugging and understanding of software.</p> <p>A prototype visual testing tool is presented and evaluated here as a proof of concept for some of the aspects of visual testing. Finally, some suggestions for future research in visual testing are presented.</p>	
<p>Keywords: visual testing, visual debugging, algorithm simulation, algorithm animation, debugging</p>	

Teknillinen korkeakoulu Diplomityön tiivistelmä

Tekijä:	Jan Lönnberg
Työn nimi:	Visual testing of software
Työn nimi suomeksi:	Visuaalinen ohjelmistotestaus
Päivämäärä:	7. lokakuuta 2003
Sivuja:	1 + x + 88
Osasto:	Tietotekniikan osasto
Professori:	T-106 (Ohjelmistotekniikka)
Pääaine:	Ohjelmistojärjestelmät
Valvoja:	Lauri Malmi
Ohjaaja:	Ari Korhonen
<p>Ohjelmistojen kehittäminen on altis kalliille ja henkilöaikaa syöville virheille. Virheet esitään yleensä suorittamalla ohjelmaa eri syöteillä ja tarkistamalla tuloksien oikeellisuus, eli <i>testaamalla</i>. Nykyiset vianetsintäohjelmat eivät riittävästi hyödynnä visualisaatio- ja vuorovaikutusmenetelmien tarjoamia mahdollisuuksia.</p> <p>Tässä diplomityössä esitetään uusi vuorovaikutteinen graafinen ohjelmistotestausmenetelmä nimeltään <i>visuaalinen testaus</i>. Visuaalisen testauksen työkalu tarjoaa ohjelmoijalle mahdollisuuden tutkia ja manipuloida ohjelmaa ja sen tietorakenteita.</p> <p>Läheisistä aihealueista (vianetsinnästä, ohjelmistovisualisoinnista ja algoritmianimaatiosta) tutkitaan järjestelmiä, jotka tarjoavat hyödyllisiä menetelmiä visuaaliseen testaukseen. Mahdollisesti hyödylliset tekniikat kuvaillaan, tutkitaan ja arvioidaan. Tämän perusteella suunnitellaan visuaalinen testaustyökalu, joka yhdistää käyttäjän ohjaaman algoritmisimulaation, korkeatasoisen tiedon visualisoinnin ja visuaalisen vianetsinnän ja tekee testaamisen, vianetsinnän ja ohjelmistojen ymmärtämisen helpommaksi.</p> <p>Tässä diplomityössä myös esitetään ja arvioidaan prototyyppi visuaalisesta testaustyökalusta, jonka tarkoitus on osoittaa osittain visuaalisen testauksen toimivuutta. Lopuksi esitetään muutama ehdotus tulevalle visuaalisen testauksen tutkimukselle.</p>	
<p>Avainsanat: visuaalinen testaus, visuaalinen vianetsintä, algoritmisimulaatio, algoritmianimaatio, vianetsintä</p>	

Tekniska högskolan

Sammandrag av diplomarbetet

Utfört av:	Jan Lönnberg
Arbetets namn:	Visual testing of software
Arbetets namn (på svenska):	Visuell testning av programvara
Datum:	7 oktober 2003
Sidantal:	1 + x + 88
Avdelning:	Avdelningen för datateknik
Professur:	T-106 (Programteknik)
Huvudämne:	Programsystem
Examinator:	Lauri Malmi
Handledare:	Ari Korhonen
<p>Då man utvecklar programvara råkar man ofta ut för fel som tar mycket tid och pengar att reda ut. Felen spåras och avlägsnas (<i>avlusning</i> eller <i>debugging</i>) vanligen genom att man utför programmet med olika input och kontrollerar resultaten, vilket kallas <i>testning</i>. De befintliga avlusningsprogrammen utnyttjar inte till fullo alla möjligheter som visualisering och interaktion erbjuder.</p> <p>I detta diplomarbete presenteras ett nytt interaktivt grafiskt testningsförfarande för programvara som kallas <i>visuell testning</i>. En programmerare kan med ett verktyg för visuell testning undersöka och manipulera ett aktivt program och dess datastrukturer.</p> <p>I diplomarbetet undersöks system i närliggande områden (avlusning, programvaruvisualisering och algoritmanimation), och potentiellt användbara tekniker som används i dessa beskrivs, undersöks och bedöms. På basen av dessa skapas en design för ett visuellt verktyg för testning av programvara. Detta verktyg kombinerar olika aspekter av användarkontrollerad algoritmsimulation, datavisualisering på hög abstraktionsnivå och visuell avlusning. Verktöget förenklar testning, avlusning och förståelse av programvara.</p> <p>En prototyp av det visuella testningsverktyget presenteras och bedöms också i detta arbete. Till slut presenteras några förslag för framtida forskning i visuell testning.</p>	
<p>Nyckelord: visuell testning, visuell avlusning, algoritmsimulation, algoritmanimation, avlusning</p>	

Acknowledgements

This thesis was written at Helsinki University of Technology as a part of the software visualisation research of the Laboratory of Information Processing Science.

I am greatly indebted to my supervisor, Professor Lauri Malmi, for much of the inspiration for this thesis and his enthusiastic guidance, without which this thesis would have been noticeably less thorough and structured.

I likewise owe much to my instructor, Ari Korhonen, for the support, suggestions and inspiration he has provided. His ideas, methods and work are reflected throughout this thesis.

I am also grateful to Panu Silvasti for his kind assistance and Markku Rontu for his helpful suggestions and comments.

Finally, I want to express my gratitude to my parents for their love, support and understanding. This thesis is dedicated to them.

Otaniemi, 7th October 2003,

Jan Lönnberg

Contents

1	Introduction	1
1.1	Goal	1
1.2	Proposed solution	2
1.3	Thesis outline	2
2	Objectives	3
2.1	Criteria	3
2.2	Scope	4
2.2.1	Target language	4
2.2.2	Solution types to consider	5
2.2.3	Prototype	5
3	Related work	6
3.1	Debugging	6
3.1.1	DDD	7
3.1.2	RetroVue	7
3.1.3	ODB	8
3.1.4	Amethyst	8
3.1.5	Lens	8
3.1.6	BlueJ	8
3.2	Program visualisation	9
3.2.1	Jeliot	9
3.2.2	VCC	9
3.2.3	UWPI	9
3.2.4	Korsh-LaFollette-Sangwan	9
3.2.5	Prosasim	10
3.2.6	Leonardo	10
3.2.7	VisiVue	10
3.2.8	Pavane	10
3.2.9	DynaLab	10
3.2.10	Tarraingím	10
3.2.11	JAVAVIS	10
3.3	Algorithm animation and simulation	11
3.3.1	Matrix	11
3.3.2	JDSL	11
3.3.3	Balsa-II	11
3.3.4	Zeus	11
3.3.5	AlgAE	12
3.3.6	World-wide algorithm animation	12
3.3.7	JAWAA	12
3.3.8	JCAT	12
3.3.9	JIVE	12

3.4	Evaluation	12
3.5	Analysis	12
3.5.1	Traditional grouping	14
3.5.2	Grouping by data extraction approach and code preprocessing	15
3.5.3	Grouping by view control style	16
3.5.4	Conclusions	17
4	Visualisation	18
4.1	Primitive value representations	18
4.1.1	Textual representation	18
4.1.2	Graphical representations	19
4.1.3	Evaluation	19
4.2	Array representations	19
4.2.1	Arrays as lists	20
4.2.2	Arrays as tables	20
4.2.3	Arrays as plots	20
4.2.4	Arrays as images	22
4.2.5	Evaluation	22
4.3	Class and object representations	23
4.3.1	Nesting or arrows?	23
4.3.2	Static fields	25
4.3.3	Labelling	25
4.3.4	Methods	25
4.3.5	Indicating the origin of a reference	25
4.3.6	Evaluation	26
4.4	Code visualisation	26
4.4.1	Displaying source code	27
4.4.2	Full-detail graphical representations of program code	27
4.4.3	Colour pixel and line views of program code	27
4.4.4	Graphical hierarchies for classes	27
4.4.5	Cross-references in code	28
4.4.6	Evaluation	29
4.5	Execution visualisation	30
4.5.1	Tracing through source code	30
4.5.2	Highlighting objects	30
4.5.3	Annotating objects	30
4.5.4	Drawing operations as connections	31
4.5.5	Viewing the execution stack	31
4.5.6	Viewing the call tree	31
4.5.7	Sequence diagrams	32
4.5.8	Collaboration diagrams	33
4.5.9	Hybrid diagrams	33
4.5.10	Assignment search	35
4.5.11	Slicing and dependence graphs	35
4.5.12	Evaluation	41
4.6	Summary	41
5	Elision and abstraction	43
5.1	Data abstraction	43
5.1.1	Identification by class or interface	44
5.1.2	Identification by patterns	45
5.1.3	Accessors and modifiers	46
5.1.4	User-defined abstraction	48
5.1.5	Combining abstractions	48

5.1.6	Data abstraction model	49
5.1.7	Evaluation	50
5.2	Data elision	50
5.2.1	Automatic elision control	51
5.2.2	Manual elision control	51
5.2.3	Evaluation	51
5.3	Code elision	52
5.3.1	Structure-based elision	52
5.3.2	Slicing	52
5.3.3	Evaluation	52
5.4	Execution elision	52
5.4.1	Call tree-based elision	53
5.4.2	Filtering	53
5.4.3	Dynamic slicing	54
5.4.4	Evaluation	54
5.5	Summary	54
6	Controlling the debuggee	55
6.1	Data modification	55
6.1.1	Textual editing	55
6.1.2	Graphical reference manipulation	55
6.1.3	Graphical primitive entry	55
6.1.4	Graphical expression entry	56
6.2	Method invocation	56
6.2.1	Textual invocation	56
6.2.2	Graphical invocation	56
6.3	Starting and stopping execution	56
6.4	Summary	57
7	Implementation	58
7.1	Connection to debuggee	58
7.1.1	Instrumentation of code before or at compilation	58
7.1.2	Instrumentation of compiled code	59
7.1.3	Instrumented interpreter	59
7.1.4	JPDA	59
7.1.5	Hybrid debuggee connection	60
7.1.6	Evaluation	62
7.2	Manipulation of program history	62
7.2.1	Animation based on logging	62
7.2.2	Reverse execution	63
7.2.3	Evaluation	63
8	Tool designs	64
9	Prototype	66
9.1	Connection to debuggee	66
9.1.1	Instrumentation	66
9.1.2	Runtime debuggee connection	68
9.2	Data model	69
9.3	View model	69
9.4	User interface	69

10 Use cases	72
10.1 Debugging a sort routine	72
10.2 Testing a hash table	73
10.3 Examining a data structure through a library API	74
10.4 Studying the behaviour of a large program	74
11 Evaluation of prototype	76
11.1 Feature set comparison	76
11.2 Evaluation using use cases	76
11.2.1 Debugging a sort routine	77
11.2.2 Testing a hash table	78
11.2.3 Examining a data structure through a library API	79
11.2.4 Studying the behaviour of a large program	80
11.2.5 Evaluation	80
11.3 Summary	81
12 Conclusion	82
12.1 Future research	83

Terminology

This section defines some terminology that is used in this thesis.

- Debugging:

Debugging Examining a program in order to find and eliminate errors.

Debuggee The program that is being examined in debugging.

Debugger A program used in debugging to examine and affect what the debuggee is doing.

- Software visualisation:

Visualisation Graphical representation of information.

Algorithm animation Algorithm visualisation using visualisations of its data structures at sequential time steps that can be traversed backwards or forwards. Sometimes referred to as *discrete animation*, as opposed to *continuous* or *smooth* animation, in which graphical objects move smoothly from one place to another.

Algorithm simulation Allowing the user to manipulate a data structure himself as if he were the algorithm. Also called *user controlled simulation of an algorithm*.

- Non-object-oriented programming:

Record/struct A set of fields.

Field A named variable belonging to a record or struct.

Procedure/function A named sequence of instructions that may take input parameters and may return a value.¹

- Object-oriented programming:

Object A set of fields and methods; an instance of a class.

Field A named variable belonging to an object.

Variable A memory location that can contain a value.

Class A definition of a type of object. The fields and methods of all instances of a class are specified in the class. A class may be a *subclass* of another class (its *superclass*), in which case all instances of the subclass are instances of the superclass. A subclass *inherits* the methods and fields of its superclass. The inherited methods may be *overridden* in the subclass. Some languages may allow a class to have multiple superclasses.

Method A procedure associated with a class or object.

Method invocation The act of executing a method.

¹The terms “procedure” and “function” mean slightly different things in different languages. Pascal procedures do not return a value, while functions do. In C, both are considered functions.

Local variable A named variable belonging to a method invocation.

Array A data structure that contains zero or more variables that are indexed by number.

- Virtual machines:

VM (Virtual Machine) A computing device simulated in software.

JVM (Java Virtual Machine) A VM that executes bytecode, as defined in [36]. Bytecode is usually generated by compiling programs written in Java, which is defined in [21].

- Data types (defined more precisely and in more detail in [30]):

ADT (Abstract Data Type) A set of operations with defined semantics. This corresponds roughly to the specification of an interface in an object-oriented program.

CDT (Conceptual Data Type) An implementation of an ADT in a programming language.

FDT (Fundamental Data Type) The static part of a CDT in which all data types are generic (i.e. the types of the values stored in the FDT are irrelevant).

Chapter 1

Introduction

As software has grown more complex, the amount of errors in it, known as *bugs*, has increased. Market pressures can further compound this problem by causing a project to be developed with unskilled programmers or insufficient time or money. It is estimated that software errors lead to costs of tens of billions of euros every year. [49]

Bugs are essentially a difference between the intended behaviour of the program and its actual behaviour. Thus, one way to find and eliminate bugs (an activity known as *debugging*) is to examine the operation of the program and compare this to the desired operation. This approach is called *testing*. Tools that assist in debugging by allowing programmers to examine the current state of a program (which includes the data the program is working with in memory and the currently executing code) and control its execution are called *debuggers*.

Current debuggers have several limitations. As they generally show data by displaying the values of individual variables, it is often hard to see the interesting aspects of the running program and its data. Object-oriented development has allowed programmers to hide unnecessary detail while developing, but debuggers generally do not take advantage of this. Furthermore, it is difficult to test results of operations in the program without writing additional code that runs parts of the program and examines the results. Also, when a problem is found, its cause is often lost in the past, which necessitates careful rerunning and stepping through the program to find the cause of the problem.

In order to teach students algorithms better, many universities have developed *algorithm animation* tools that display the execution of algorithms as a sequence of graphical representations of a data structure. Algorithm animation can also be used in conjunction with user-controlled *algorithm simulation*. Algorithm simulation allows students to examine the behaviour of algorithms by specifying the operations to perform and watching the results. Usually, the algorithm simulation tool provides a graphical user interface (*GUI*) that shows the data structures and allows the user to perform operations on them (such as adding or modifying data) using common GUI input techniques such as clicking or dragging and dropping.

Algorithm animation and simulation tools provide a way of visualising data structures that makes the relevant data easier to find and comprehend and a simple mechanism for controlling operations on these data structures. User-controlled algorithm simulation is conceptually quite similar to testing, which suggests that some techniques used in algorithm simulation can be applied to testing.

1.1 Goal

It seems that an unfulfilled need for a better way to examine and test software exists. Specifically, something is needed to aid in the following tasks:

- Testing code to see if it works and identifying the faults if it doesn't.
- Studying code to understand what it does and how it works.

1.2 Proposed solution

The solution presented here to the problem of testing and examining programs is *visual testing*, in which the visualisation and control techniques of algorithm simulation are applied to the problem of testing software. The programmer using visual testing should be able to examine the operation of a program visually without being bogged down with implementation details. He should also be able to choose which parts of the program to execute and manipulate the data to be processed by the executing program. The visual testing tool should interact with its users through a graphical user interface.

1.3 Thesis outline

The following issues are addressed in this thesis:

- The desired properties of a visual testing tool (Chapter 2).
- A survey and evaluation of debugging and software visualisation tools that can be used for visual testing (Chapter 3).
- Descriptions and evaluations of visualisation (Chapter 4), elision and abstraction (Chapter 5) and control techniques (Chapter 6) suitable for visual testing.
- Different approaches to the implementation of a visual testing tool (Chapter 7).
- The design of a fully fledged visual testing tool (in Chapter 8).
- The design and implementation of a prototype visual testing tool that demonstrates the feasibility of the visual testing concept and the new techniques applied in it (in Chapters 8 and 9).
- Some use cases that can be used to demonstrate and evaluate the prototype visual testing tool (Chapter 10).
- An evaluation of the prototype (Chapter 11).
- Conclusions and suggestions for future studies (Chapter 12).

Chapter 2

Objectives

The goal of a visual testing tool is to provide programmers with the ability to examine what their program does interactively, by allowing monitoring and manipulation of program execution and data. This allows the programmer to try out the results of manipulating objects and executing methods.

2.1 Criteria

The goal of a visual testing tool can be split into the following criteria:

Generality The testing tool should work on programs not specifically designed or written for visualisation; in other words, the user should not need to change his programs to use the tool with them. Ideally, any program can be examined. This criterion corresponds to requiring *generality* and *scalability* (both parts of *scope*) as defined by Price et al. in their taxonomy of software visualisation [46]. Lack of generality usually means that programs must be rewritten to fit the tool. However, dissimilar programming languages require different visualisation strategies, so it is unrealistic and not very useful to be able to use the same tool on programs written in completely different languages.

Completeness All aspects of the running program should be accessible for examination. In most object-oriented languages this encompasses:

- An *execution stack* or *call stack* for each execution thread, which typically contains information on the currently active method invocations and their local variables.
- All objects and variables.
- Code and current execution position.

This corresponds to *fidelity and completeness* in the Price et al. taxonomy.

Data modification Variable values should be freely modifiable where allowed by the programming language. Software visualisation does not usually address this aspect, although it is common in debuggers.

Execution control The user should be able to try out any part of the program on data of his choice and execute operations of his choice on the data in the running program. Ideally, the user should be able to control what is executed down to individual operations and create and modify classes and methods on the fly. Being able to invoke methods at will is the most important part of execution control. Control of this type is a fundamental part of algorithm simulation.

Presentation Ideally, the testing tool would automatically present exactly what the user wants to know about the program, its execution and its data in the form in which he thinks about these matters. The data should be represented in such a way that the user can easily find the information he requires, and the information is expressed clearly. Visual representation is clearly the most effective way of conveying this information in practice, although visual information can be augmented with sound or information for any other sense. Finding a visual representation that meets these requirements is one of the main problems in constructing a visual testing tool. In practice, this includes:

Representation The data must be shown in a suitable (graphical) form.

Abstraction Unnecessary implementation details should be abstracted away whenever it is possible and the user desires it. Implementation details that were hidden from the user during programming that he does not care about should also be hidden while debugging (one aspect of *appropriateness and clarity* in Price et al.).

Automatic view control The tool should guess at what the user wishes to see and present the data in this form (another aspect of *appropriateness and clarity* in Price et al.).

Manual view control The user should be able to change the view easily to match his own ideas by hiding (*eliding*) parts of the view and changing the way in which information is represented (*navigation* in Price et al.).

The mapping between the data in the running program and the visualisation should work both ways; if the user modifies data in the graphical view, the corresponding change should be made to the data in the debuggee.

Causal understanding It should be easy to understand the reason for the current state of the program. Ideally, we could ask the computer something like “Why is this reference `null`?”, and it would answer, for example, “Because you put these two statements in the wrong order.”. In order to do this, the computer would have to understand what was expected of it and be able to write correct code, which would defeat the purpose of having an interactive testing tool. Therefore, a more realistic goal must be set.

A more realistic goal is that the user should be able to examine the entire execution history of the program (or at least selected interesting parts of it) and search through it for the instructions, instruction sequences or events that caused a specified change to the state of the program or caused it to differ from the expected state. This is the approach taken by algorithm animation.

2.2 Scope

Designing and writing a visual testing tool that provides all features that could possibly be useful for the testing, debugging or examination of any program written in any language is a very large undertaking. Therefore, the scope of this thesis must be a carefully bounded small subset of the area of visual testing.

2.2.1 Target language

Object-oriented languages are especially well suited to working at high abstraction levels, as they provide many mechanisms for encapsulation and abstraction. Many languages that are in heavy use today are object-oriented, such as Java and C++. Because of these two factors, this thesis will focus on object-oriented languages.

The prototype is designed to process programs written in Java (described in [21]), because:

- Java is highly portable, as Java software can be run on any platform with a JVM. This makes the results more widely applicable.
- Java is object-oriented, which encourages programmers to write code at a higher abstraction level and with more modularity than in e.g. C. This makes it easier to construct visualisations of a program similar to the concepts in a programmer's mind.
- Java is widely used:
 - A lot of software has been written in Java.
 - Java is often used to teach programming.
 - Programmers with Java skills are highly sought after [76, 79].

This means that a wide range of software is available and being written in Java that can be used with the results of this work.

- Java is considerably less complex than C++. This simplifies the design a lot.

In matters that are not specific to Java, generalisation to other similar languages is also discussed here.

2.2.2 Solution types to consider

The solutions to the problems posed in the introduction will be based on techniques from existing debugging, software visualisation and algorithm animation and simulation software, whenever these techniques can be adapted to suit visual testing.

Techniques that seem to provide little extra comprehensibility but require a lot of extra work, such as smooth animation, sound or three-dimensional visualisation, will also be left out of consideration. This work will concentrate on ways to visualise the data structures and execution flow of a running program using two-dimensional graphics and interact with the running program.

2.2.3 Prototype

In order to examine how well visual testing works in practice, a working implementation should be made. However, features that are not essential to visual testing can be left out, especially if other software already provides similar functionality. This prototype should allow examination of programs written in Java. If possible, existing software (debuggers, visualisers or similar tools) will be used as a basis for the prototype.

Chapter 3

Related work

Software visualisation can be divided into three categories based on the purpose of the visualisation and the approaches used:

- Visual debugging.
- Program visualisation.
- Algorithm animation and simulation.

The program visualisation and debugging approaches are based on the idea of taking a running program, stepping through it and showing the variables and other interesting parts of the state of the executing program. This shows what the program is doing. Program visualisation is used to understand a program, while debugging involves finding and correcting errors in a program. Program visualisation and debugging can be done using similar techniques and in some cases both may be combined in a single tool (e.g. Lens [37, 38] and Leonardo [23]).

The algorithm animation approach is based on showing a series of graphical representations of a data structure at successive points in time in order to explain how an algorithm operates on the data structure. Many algorithm animation systems allow the user to step back and forth through the states of the data structure and algorithm to study the progress of the algorithm. User-controlled algorithm simulation gives the user the ability to manipulate the data structures himself.

In this chapter, debugging and program visualisation systems designed for procedural languages (with or without object-orientation) such as C, C++, Pascal and Java are surveyed. Debuggers and program visualisers for other languages are only included if they have features or properties that may be of use in visual testing; visualising the execution and data structures of programs written in e.g. Prolog or Lisp is an entirely different problem due to the different execution and data models used in these languages. Also, the most important general-purpose algorithm animation and simulation tools are described.

3.1 Debugging

Debuggers are intended to be used to find bugs in programs by tracing through the execution of program and examining and editing variable values. Most debuggers have some execution control facilities (e.g. single step, breakpoints, watchpoints and expression evaluation) and some way to view variable values.

Most debuggers are based on ideas from *FLIT* (*Flexowriter Interrogation Tape*), which introduced symbolic debugging (meaning that the user could work with variable names, labels and instruction mnemonics instead of memory addresses and numerical instruction codes) and breakpoints [55]. More than 40 years later, most current debuggers are based

on the same paradigm: run the program to a specified point, stop it and examine the values of the variables. A few improvements have been made, such as single stepping. These debuggers can be referred to as *command line debuggers* (referring to their user interface) or *traditional debuggers* (referring to their heritage).

The most common style of debugger today is a traditional debugger with a graphical user interface. Most *integrated development environments* (development software packages containing an editor, a compiler and linker and a debugger within a common user interface), such as Borland JBuilder and Microsoft Visual Studio, contain a debugger of this type [72, 80]. Debuggers of this type are referred to by their authors as “visual debuggers” (e.g. the Javix Visual Debugger [68] or the Tango/04 VISUAL Debugger [78]) or “graphical debuggers” (e.g. KDbg [75] or JSwat [66]). To avoid confusion, I will refer to debuggers of this type as *graphical debuggers*. These debuggers have few interesting features from a visual testing viewpoint and are too numerous to survey properly. For these reasons, they are not included here unless they have other features that merit attention.

During the last two decades, visualisation features have been developed for debugging tools. Two approaches to visualisation in debuggers can be discerned. The simpler approach is to allow the user to select concrete data structures to be displayed and display the primitive values in them and references between them without any attempt to interpret the meaning of the data. This approach is used by most debuggers intended for development use (e.g. DDD [63] and GVD [67]) and those intended for novice programmers (e.g. Amethyst [39]). The other approach allows the user to create visualisations for his program by constructing animations using predefined primitives and the variables in the program. This allows the user to view his data at a higher level of abstraction, but also requires more work to get the desired view. This approach is also sometimes used in program visualisation, which makes it hard to classify some of the programs that use it (e.g. Lens [37, 38]). Debuggers that use software visualisation techniques are generally called *visual debuggers* although they are also sometimes confusingly referred to as “graphical debuggers”.

BlueJ [27] is not quite a debugger; it is primarily a development and testing tool, although it also has debugging features. It is included here because of its interesting approach to testing.

In the following, each surveyed debugger is briefly presented.

3.1.1 DDD

DDD (Data Display Debugger) [62, 63] is a visual debugger that provides extensive debugging facilities and some (low-level) data structure visualisation, mostly limited to displaying structs or classes as boxes with pointers shown as arrows between them, as well as plotting array data using the plotting program *gnuplot*. DDD makes use of a command line debugger, such as GDB or JDB, to debug programs written in a variety of languages, such as C, C++, Java, Pascal, FORTRAN, Python and Perl.

GVD (GNU Visual Debugger) [67] has a similar feature set and user interface.

3.1.2 RetroVue

RetroVue [14, 81] by VisiComp allows the user to browse the execution history of an executing Java program by watching it execute (either in real time or from a log), by stepping backwards or forwards through the logged states or by searching for specific events. The state of the program is shown using the following views:

- A thread view that shows the state of the different threads (running, runnable, blocked, et.c.) as a function of elapsed time. Locking and deadlocks are clearly indicated.
- The execution history of the program (as a tree structure of nested method calls and statements).

- A tree view of the static structure of the program (classes, fields, methods, et.c.).
- A tree-structured data view based on showing the local variables and expanding branches representing references to other objects.

RetroVue is thus essentially a graphical debugger with the ability to step back and forth through the execution history of a program or examine the history as a tree.

3.1.3 ODB

Like RetroVue, *ODB* (the “Omniscient Debugger”) [35] collects information about the operations performed by an executing Java program and allows the programmer to examine the execution history. Like RetroVue, *ODB* supports stepping backwards and forwards through the execution history. Unlike RetroVue, *ODB* allows the user to interrupt the running program and execute methods and modify data values in a secondary timeline that starts from a copy of a state of the real execution history. The main timeline containing the real execution of the program may not be modified.

ODB shows the threads in the program, their stack, the tree of executed methods and a treelike view of selected objects and objects they refer to.

3.1.4 Amethyst

The *Amethyst* visual debugger [39] displays call stacks, variables, arrays and records graphically for a running Pascal program. Records and arrays are displayed as nested boxes. Heap objects and pointers are not supported. *Amethyst* also provides graphical control over stepping and breakpoints.

3.1.5 Lens

The *Lens* visual debugger [37, 38] is an attempt to bridge the gap between program visualisation and algorithm animation. It is based on the *XTango* animation system and the *dbx* command line debugger, and allows the construction of animations based on data in programs. The user constructs an animation by creating graphical objects (lines, rectangles, text and object arrays, for instance) and adding animation commands that affect these objects to the source code, such as “move”, “colour” or “delete”. The actions are defined using a graphical editor. *Lens* has some limited execution control facilities and allows the user to access the underlying debugger directly.

Getting a high-level visualisation out of *Lens* requires quite a lot of extra work, as you essentially have to design the visualisation yourself. The additional programming required to produce a visualisation discourages programmers from using *Lens*. [24]

3.1.6 BlueJ

BlueJ [27] is an integrated development environment for Java designed for use in introductory programming courses. *BlueJ* displays the structure of a Java program in a fashion similar to a UML class diagram, and allows the user to graphically instantiate objects and execute methods. *BlueJ* allows the user to inspect values of variables, but it does not provide any data visualisation beyond a simple graphical debugger, which supports stepping, breakpoints and can show lists of local variables, fields of an object or class and currently active methods.

3.2 Program visualisation

The purpose of program visualisation is primarily educational. The goal is usually to help the user understand or explain to others how a program works. Most of these systems are targeted at teaching basic programming (especially Eliot [58], its successor Jeliot [22] and the system designed by Korsh, LaFollette and Sangwan [32, 33, 50]), while others are (also) intended to help programmers understand what a program is doing (e.g. VisiVue [81] and Prosim [73]).

While debuggers are traditionally based on stopping program execution and examining the state of the program, program visualisation systems use a greater variety of approaches to extracting information from a running program. Some require that the user add visualisation commands to their program (e.g. Leonardo [16]), while others automatically add visualisation code to the user's program using a modified compiler or additional precompiler (e.g. Eliot [58] and Jeliot [22], VCC [5] and UWPI [23]). Finally, some use debugger-style examination of running programs (e.g. VisiVue [81]).

While most program visualisers use a finished program (possibly with graphics calls added) as input, Prosim [73] is based on simulating a system described as a UML model.

Program visualisers with no features relevant to visual testing that cannot be found in other program visualisers have been left out. These include the system described by Rasala in [47]. Systems that only visualise source code or other static structures, such as Source-Navigator [77] are not included here.

3.2.1 Jeliot

The *Eliot* [58] and *Jeliot* [22] program visualisers display the data structures of an executing program (at quite a low level; primitives, arrays, stacks and queues) with smooth animation by instrumenting the code on compilation. Jeliot is used as a client/server program over the web; the client supplies source code in EJava (a modified version of Java with added stack and queue types and some limitations), which the server precompiles to Java (adding animation code in the process) and compiles into an animation applet. Eliot uses C and C++ instead of Java and is not designed for use over WWW. Eliot also works with low-level built-in data types such as integers, arrays and trees.

3.2.2 VCC

The *VCC* [5] system adds animation features to C programs using a modified compiler that adds animation code. VCC shows the currently active function (with arguments and local variables), the tree of executed program calls, the program code, standard I/O and separate data views for records (structs) and arrays. However, VCC does not visualise dynamic (heap-allocated) structures.

3.2.3 UWPI

The *UWPI* [23] system is based on a specialised Pascal compiler that adds data visualisation that attempts to recognise known idioms (common data structure operations) and from this recognise abstract data structures for visualisation such as Boolean or reference variables.

3.2.4 Korsh-LaFollette-Sangwan

The system designed by Korsh, LaFollette and Sangwan [32, 33, 50] is essentially a data visualisation and animation system for C/C++ programs based on modified data types with overloaded operators containing animation calls. It displays the code, heap, call stack, local variables, arguments and operations being performed. The system is intended for use in basic programming courses and therefore only handles integers, structs and pointers.

3.2.5 Prosim

Prosim [73] takes an executable UML model of a program built using the *Prosa* modeller and simulates its execution. The model can then be visualised using the UML diagrams (e.g. collaboration diagrams) in the model. The values of attributes in the model can also be examined. Using the *Prosaj* or *Prosacpp* code generators, this model can be converted into an executable model in Java, C or C++.

3.2.6 Leonardo

The *Leonardo* [16] software visualisation environment allows the user to edit, compile, execute and animate C programs. It uses a virtual processor to provide debugging facilities including reverse execution. Graphical interpretations are specified using declarations written in the logic programming language *Alpha* embedded in the C program as comments. Using *Alpha*, the programmer can construct many types of visualisations containing geometric primitives or graphs. However, *Leonardo* is hard to classify, as it combines aspects of emulation, debugging and program visualisation.

3.2.7 VisiVue

VisiVue [81] by VisiComp visualises and animates objects in an executing Java program. The animation is done while the program executes, with highlighting to indicate the currently executing statement. It also produces textual execution trace logs.

3.2.8 Pavane

Pavane [48] visualises the state of a program written in *Swarm*, consisting of a set of transition rules and a defined initial state to which the rules are applied. *Pavane* uses declarative visualisation; a mapping between the program and a world of 3D geometric objects is defined as a set of rules.

3.2.9 DynaLab

DynaLab [9] consists of a virtual machine connected to a simple program animator that displays the current execution position in the source code, the currently active procedures and their local variables textually. The virtual machine is capable of reverse execution. Compilers for *DynaLab*'s virtual machine exist for Pascal, Ada, C and C++. Only the Pascal compiler has been completed and released.

3.2.10 Tarraingím

Tarraingím [40] visualises programs written in the object-oriented programming language *Self*. Besides displaying the actual data contents of objects, *Tarraingím* can display objects graphically at a higher level of abstraction using view code written to monitor and access objects through their interfaces.

3.2.11 JAVAVIS

JAVAVIS [43] visualises the current state of a Java program as a set of UML object diagrams (one for every active method invocation) containing the local variables of the method and all objects reachable from these local variables by following references. *JAVAVIS* also visualises the executed method calls of a Java program as a sequence diagram.

3.3 Algorithm animation and simulation

Most algorithm animation and simulation tools are designed for the teaching of algorithms and data structures. Their purpose is twofold: to make it easier for teachers to show their students what an algorithm does (Balsa-II and Zeus emphasise this application [10, 11]), and to allow the student to experiment with data structures and algorithms (Matrix and JDSL emphasise this area [6, 29, 31]). Algorithm animation tools designed for students' use often automatically visualise data structures that conform to a predefined interface, while those designed for teachers' needs usually require the user to add explicit graphics calls.

This survey does not include graphics libraries that only provide geometric primitives and animation, such as XTango [52] and Polka [54]. These libraries leave most of the hard work of constructing a visualisation to the user. Animation tools that only work with geometric primitives, such as ANIM [8] and Samba [53] and its derivatives, have been left out for similar reasons. Systems superseded by newer systems by the same authors, such as Balsa [13] and WWW-TRAKLA [28], have also been left out. Algorithm animation systems designed for a few specific algorithms or a small class of algorithms (e.g. geometric algorithms) have been left out of consideration due to their amount and limited applicability. A wide range of specialised algorithm animation tools can be found at [65].

3.3.1 Matrix

The *Matrix* [29, 31] system provides animation (including stepping both backwards and forwards) and user-controlled simulation of data structures written to conform to specified interfaces. Unlike the other systems described here, Matrix allows hierarchical composition of types. Matrix supersedes the old *Trakla* system, which was limited to user-controlled simulation of a few built-in data structures.

3.3.2 JDSL

The *JDSL Visualizer* [6] provides animation and visualisation of data structures written to conform to specified interfaces (those of the JDSL data structure library). By default, it shows the data structure before and after API calls, but additional animation frames can be generated by adding calls to the visualiser. The JDSL Visualizer allows the user to select methods and their arguments and execute the methods (as defined by the user) on the data structures. JDSL shows the history of events that have happened to the data structure and the current state of the data structure.

3.3.3 Balsa-II

The *Balsa-II* [10] system is an algorithm animation tool. It animates algorithms written in Pascal with explicitly added display calls and conforming to a specified interface. The algorithm outputs change events through an adapter to a modeller, which maintains a generic data model that can be used by several viewers, which display the data in the model.

3.3.4 Zeus

The *Zeus* [11] algorithm animation system is similar to the authors' previous system, Balsa-II, but adds support for allowing the user to generate events with specified arguments (similar to calling methods). Zeus works with Modula-2 code.

3.3.5 AlgAE

AlgAE [61] animates algorithms that are implemented in Java or C++ conforming to the visualiser's interfaces. The algorithms must be annotated with explicit visualiser calls. AlgAE also provides a graphical user interface with which the user can invoke algorithms on a data structure. The visualisation consists of boxes that can contain text, other boxes and links or arrows to other objects. AlgAE does not appear to support moving back and forth through the execution of the algorithm.

3.3.6 World-wide algorithm animation

The *World-wide algorithm animation* [25] system (hereinafter *WWAA*) is designed to allow students to manipulate (through a web browser) algorithm implementations written in Pascal (with lots of calls to the animation system) running on a server. WWAA allows users to step through an algorithm implementation or run it to the end or a breakpoint and view and modify variable values. It can also forward bitmap images containing graphical representations of data structures.

3.3.7 JAWAA

JAWAA [44] executes animation scripts generated by a program to which animation output commands have been manually added. The animation can include primitive graphical objects such as lines, text and rectangles as well as arrays, stacks, queues, graphs and trees. Once the animation script has been created, the user can run or step forward through the animations.

3.3.8 JCAT

JCAT [12] animates algorithms written in Java annotated with visualiser calls. The visualiser calls are passed to a view applet designed for a specific data structure or algorithm. The view applet uses an animation package based on a graph containing vertices that can be connected with edges and moved to different positions smoothly. The vertices can have various graphical properties such as a textual label, a polygonal outline or colour.

3.3.9 JIVE

JIVE [70] animates algorithms written in Java using a set of pre-written data structures with animation hooks. The data structures supported by JIVE include graphs, binary trees, lists and hash tables. The algorithm can request user input, such as selecting a graph vertex or entering a number. JIVE also allows the user to manipulate the data structures graphically.

3.4 Evaluation

I have evaluated tools for suitability for visual testing by checking how well they meet the criteria mentioned in this section. The evaluation results use the notation defined in Table 3.1.

In order to evaluate the previously done work in this area, I compare the systems against the requirements listed in section 2.1. The results of this comparison are shown in Table 3.2.

3.5 Analysis

This section summarises the survey results and describes some commonalities in the surveyed tools.

Score	Meaning
-	The system does not meet the criterion at all; the system has no functionality of this type.
I	The system meets the criterion partially; the system has some limited functionality of this type that may be occasionally useful.
II	The system meets the criterion well enough for basic use; the system provides functionality of this type that is usually sufficient.
III	The system meets the criterion very well; the system provides excellent functionality of this type that handles even complex cases well.

Table 3.1: Scoring system for evaluation of tools

System	Generality	Completeness	Data modification	Execution control	Representation	Abstraction	Automatic view control	Manual view control	Causal understanding
DDD	III	III	II	II	II	-	I	II	I
RetroVue	III	III	-	I	I	-	-	II	III
ODB	III	III	II	I	I	I	-	II	III
Amethyst	II	I	I	II	II	-	I	-	-
Lens	III	III	I	I	II	II	-	II	I
BlueJ	III	III	I	III	I	I	-	II	-
Jeliot	II	I	-	I	II	I	-	II	-
VCC	III	I	-	I	II	I	II	I	I
UWPI	I	I	-	-	II	II	II	-	-
Korsh et al.	II	I	-	I	II	II	II	-	-
Prosim	I	II	III	III	III	I	II	II	I
Leonardo	II	II	-	I	III	III	-	II	II
VisiVue	III	I	-	I	II	-	II	I	-
Pavane	I	II	-	I	III	III	-	III	-
DynaLab	II	II	-	II	I	-	II	-	II
Tarraingim	I	II	I	I	III	III	II	III	-
JAVAVIS	III	II	-	I	II	-	II	I	-
Matrix	I	I	II	II	III	II	II	II	II
JDSL	I	I	II	II	II	II	II	-	II
Balsa-II	I	I	-	I	II	II	II	II	I
Zeus	I	I	-	II	II	II	II	II	I
AlgAE	I	I	-	II	II	II	-	II	-
WWAA	I	I	II	II	II	II	-	II	-
JAWAA	II	II	-	I	II	II	-	III	-
JCAT	II	II	I	II	II	II	-	III	-
JIVE	I	I	II	II	II	II	II	II	-

Table 3.2: Evaluation of previous work

3.5.1 Traditional grouping

The common properties of the tools in each group and the differences between them are described in this subsection.

Debugging

In general, debuggers concentrate on generality, completeness and data modification. Execution control in debuggers is often limited to placing breakpoints and stepping (RetroVue, ODB and Lens), but some allow the user to invoke methods, functions or similar constructs (DDD, Amethyst, BlueJ).

RetroVue, ODB and BlueJ show data structures using techniques from (non-visual) graphical debuggers; they do not provide data structure visualisation. They are included here because of other interesting features; RetroVue and ODB are designed to address the problem of causal understanding, while BlueJ provides a new form of object-oriented execution control. In contrast, DDD, Lens and Amethyst all provide simple low-level visualisations.

Program visualisation

Program visualisation tools generally concentrate on presentation, although some (e.g. Leonardo) handle most of the other aspects as well. Most of these systems place much of the burden of extracting relevant information on the user (for example, Leonardo often requires extensive visualisation declarations to produce visualisations, even though the original code can be left mostly unmodified), while others ignore this problem entirely and display data structures at a very low level (e.g. Jeliot). Some program visualisers concentrate on displaying only specific types of data stored in variables (e.g. Jeliot, UWPI) or only objects (VisiVue). VCC and the system by Korsh et al. limit their support for primitives to integers, but support arrays, structs and pointers. Program visualisers often place more constraints on the program to visualise than debuggers, such as requiring programs to be written in modified versions of a common programming language (Jeliot, VCC, Korsh et al.), for a limited environment (Leonardo), in a limited subset of a language (UWPI) or in a specialised language (e.g. Pavane). In short, program visualisation tools usually concentrate on visualising a particular aspect of a program, and usually require more user intervention to produce a visualisation than a visual debugger.

Prosim is a bit of an odd man out, as it relies heavily on programs being written using the Prosa modeller. However, this means that the program can be both written and debugged using the same representations and metaphors.

Algorithm animation

Algorithm animation systems generally concentrate on presentation, abstraction and/or causal understanding. User-controlled algorithm simulation adds extensive execution control and data modification abilities to this.

The algorithm animation systems mentioned here can be used to visualise data structures in user programs, but extensive writing of code to map the data to the data types supported by the system is usually necessary. Matrix has greater expressiveness in its data structure representations than the other algorithm animation systems thanks to its ability to form nested structures inside other structures. When using most of the algorithm animation tools, the user must extensively modify his program to conform to an interface that can be visualised.

Some algorithm animation tools (e.g. Matrix and JDSL) also support algorithm simulation, which allows data structures to be modified according to data structure-specific rules.

3.5.2 Grouping by data extraction approach and code preprocessing

From the point of view of visual testing, the difference between algorithm animation, program visualisation and visual debugging is quite small. One of the most important questions is “How much do I have to modify or annotate my program to visualise it?”. The generality value shown in Table 3.2 reflects this, as the amount of changes that must be made to a program depends on the requirements placed by the visualisation system on the program. The requirements in turn depend on the approaches used to define the visualisation and monitor the state of the program.

The surveyed visual debuggers (with the exception of Amethyst, which only supports a subset of Pascal) can visualise every important aspect of practically any program written in at least one common programming language and therefore have very good (iii) generality and completeness. The visual debuggers based on taking snapshots of data at breakpoints (DDD, Amethyst, BlueJ and Lens) have very limited support for examining the execution history of a program, which implies limited (- or i) causal understanding. However, debuggers based on automatic instrumentation of Java bytecode (RetroVue and ODB) can record the execution history of a program and allow the user to examine it in many ways, which is very good (iii) for causal understanding.

Most algorithm animation and simulation tools require that a program be written to conform to their data structure or algorithm interfaces or use their predefined data structures. Rewriting a large program to conform to these interfaces can be laborious and may result in even more bugs to track down, especially if the data structures used by the visualisation differ significantly from those in the program to be visualised. Similarly, no aspects of the program’s function other than those explicitly defined in the predefined data structures or interfaces are visualised. These systems therefore have only limited (i) generality and completeness. JCAT and JAWAA only require visualisation calls or output commands to be added, which removes the need for restructuring and allows information that is not stored in a suitable data structure to be visualised. This increases the generality and completeness to a sufficient level for basic use (ii). All of these data collection techniques allow logging of execution history, but only a few of the systems actually implement this (Matrix and JDSL have a general logging facility, while Balsa-II and Zeus can provide specialised history views in some cases).

The program visualisers have the greatest variety of approaches to defining the visualisation and monitoring the program. VCC, VisiVue and JAVAVIS require almost no manual intervention to visualise almost any program written in the right language and therefore have very high (iii) generality. UWPI only supports a small subset of Pascal, which limits (i) its generality. Jeliot does not require manual modifications, but it can only visualise simple programs without modifications due to implementation limitations, which decreases its generality somewhat (ii). Leonardo and DynaLab use virtual machines and special compilers that lack some commonly used features that are available in many of the environments for which software is written, such as networking, which decreases their generality rating similarly (ii). Tarraingím, Pavane and Prosim rely on special features of unusual programming languages, meaning that most programs will have to be rewritten completely to be used with these tools, which means that they have very little generality (i). The system by Korsh et al. relies on using specialised data types, like most algorithm animation systems. However, replacing standard C/C++ data structures with those used by the Korsh system is reasonably straightforward, which means that it has sufficient (ii) generality. Most of the program visualisers concentrate on particular aspects of a program (which severely limits (i) their completeness), but some provide almost complete views of most of the interesting aspects of the program’s state (sufficient (ii) completeness).

In other words, grouping the surveyed systems by the data extraction approach and code preprocessing techniques used produces the division in Table 3.3. The scores shown in the table are the highest in each category, as this reflects the potential of the approach.

Based on this, automatic instrumentation appears to be the most suitable approach for

Type of system	Generality	Completeness	Causal understanding	Systems
Snapshots taken at breakpoints				DDD, Amethyst, BlueJ, Lens
Automatic instrumentation				RetroVue, ODB, VCC, VisiVue, JAVAVIS, Jeliot, UWPI
Predefined data structures or interfaces				Matrix, JDSL, Balsa-II, Zeus, AlgAE, WWAA, JIVE, Korsh
Annotation without restructuring				Balsa-II, Zeus, JCAT, JAWAA
Specialised execution environments				Tarraingím, Pavane, Prosasim, Leonardo, DynaLab

Table 3.3: Surveyed systems grouped by data extraction approach

visual testing.

3.5.3 Grouping by view control style

Two of the other most important questions when evaluating suitability for visual testing are “How much does the representation look like what I want?” and “How much of the work in producing the visualisation is done automatically?”. In Table 3.2, the first of these questions was answered by the representation, abstraction and manual view control ratings, while the second is answered by the automatic view control rating.

All of the debuggers (except Lens and Amethyst) and some program visualisers (JAVAVIS, DynaLab and VisiVue, Jeliot) display objects and similar data structures only if the user explicitly asks to see them as a set of fields or with minimal abstraction of implementation details. This provides sufficiently good (||) manual view control, but automatic view control that is limited at best (- or |) and a limited degree of abstraction at best (- or |). The view can consist of an object graph (which is usually a sufficiently good (||) representation) or textual field displays (which is not a clear representation in the general case (|)). Amethyst is similar, but it displays all variables in the current procedure or function (limited (|) automatic view control and no (-) manual view control). VCC provides acceptable (||) automatic view control, but is otherwise similar to this group.

The system by Korsh et al. and UWPI attempt to deduce the right abstractions and representations themselves, giving them good enough (||) support for abstraction, representation and automatic view control but no (-) manual view control.

The algorithm animation and simulation and program visualisation systems that are based on predefined data structures or interfaces that describe data structures can usually produce views that are suitable for the data structures (good or very good (|| or |||) representation and good enough (||) abstraction and automatic view control). These include Matrix, JDSL, Balsa-II, Zeus, JIVE, Tarraingím and Prosasim.

Some systems require the user to explicitly define almost every aspect of the visualisation including the position of every object in the visualisation. These are JCAT, JAWAA, WWAA, AlgAE, Jeliot, Leonardo, Pavane and Lens. They have no automatic view control, but good or very good (|| or |||) abstraction, manual view control and representation. Tarraingím provides this ability, but also works with data structures through their interfaces.

Grouping the surveyed systems by the data extraction approach and code preprocess-

Type of system	Abstraction	Representation	Automatic view control	Manual view control	Systems
Low-level visualisation with manual control	I	II	II	II	DDD, RetroVue, ODB, BlueJ, VisiVue, DynaLab, JavaVIS, VCC
Low-level visualisation with no control	-	II	I	-	Amethyst
Data structure-based visualisation	II	III	II	II	Matrix, JDSL, Balsa-II, Zeus, JIVE, Prosasim, Tarraingím
Manual view definition	III	III	-	III	JCAT, JAWAA, WWAA, AlgAE, Jeliot, Leonardo, Pavane, Tarraingím, Lens
Abstraction deduced by system	II	II	II	-	Korsh, UWPI

Table 3.4: Surveyed systems grouped by view control style

ing techniques used produces the division in Table 3.4. The scores shown in the table are again the highest in each category (Tarraingím has not been included in the automatic view control rating for manual view definition systems, as it scores highly in this category due to its use of data structure interfaces).

Clearly, basing the visualisation on specified data structures interfaces is the most suitable approach for visual testing, although adding some manual view control and automatically deduced abstractions may prove useful.

3.5.4 Conclusions

Based on the fact that each of the requirements for a visual testing tool is met by at least one of the existing systems, it seems profitable to try to combine aspects of all of these systems to produce a more useful tool.

In particular, combining the convenience of automatic instrumentation with a visualisation based on data structures, including automatic abstraction and some manual view control, could result in a system that is a lot more suitable for visual testing than any of the surveyed systems.

Chapter 4

Visualisation

This chapter examines possible approaches to various aspects of visualising a running program. The design choices made in visualisation directly affect completeness, representation and causal understanding. They may also indirectly affect manual and automatic view control.

For completeness, some sort of visualisation must be provided for every type of data and code. Completeness in this case can be seen as providing at least satisfactory visualisation for every type of data and code.

I have evaluated the visualisations described in this chapter for suitability for visual testing by checking how well they meet the criteria mentioned in this section for different types of information. The evaluation results use the notation defined in Table 4.1.

4.1 Primitive value representations

Programming languages usually contain a few primitive data types. Typically, these include several forms of real numbers and integers as well as characters and strings. This section describes some ways of displaying them.

4.1.1 Textual representation

In most cases, the best and most common way to represent a primitive value is as a character string. For example, integers have several well-known character string forms. The most common form by far is a version of the Arabic numeral system which has been in use for over a thousand years with only minor cosmetic changes [42]. This notation has later been generalised to real and complex numbers. Moreover, characters and character strings can

Score	Meaning
-	The visualisation does not visualise this type of information at all.
I	The visualisation meets the criterion partially; the visualisation can be used, but it is quite unpractical (representation); the visualisation shows a small part of the information (completeness).
II	The visualisation meets the criterion well enough for basic use; the visualisation is reasonably clear in most cases (representation); the visualisation shows most of the information (completeness).
III	The system meets the criterion very well; the visualisation is very clear (representation); the visualisation shows all the information (completeness).

Table 4.1: Scoring system for evaluation of visualisations

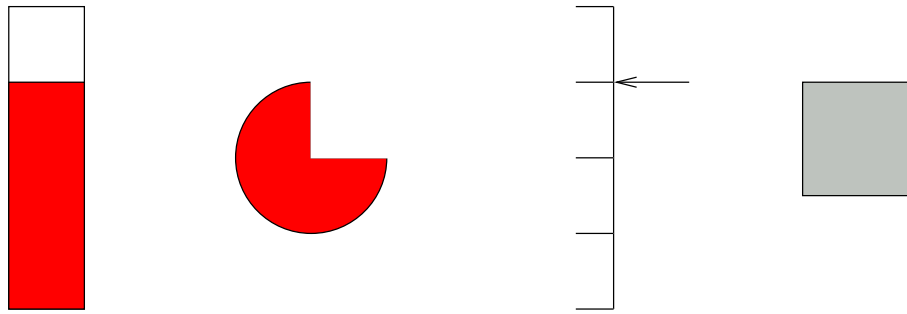


Figure 4.1: Examples of graphical representations of the real value 0.75 (from left to right: bar, circle segment, arrow position on scale, luminance)

Technique	Representation
	Primitives
Textual primitives	
Graphical primitives	

Table 4.2: Primitive representations

obviously be represented as character strings. The string representation for most primitive data types is quite compact, making textual representation a good choice for visualisation of single primitives.

4.1.2 Graphical representations

Alternative graphical representations are available for primitive types, but they are often specialised. For example, a real number that is known to be limited to a specified range can be expressed as a suitably sized bar. As most of these representations require that additional limits be placed on the values, they are unpractical for automatically generated visualisations. Program visualisation tools that require the user to define a visualisation in terms of graphical primitives (e.g. Leonardo [16]) usually support this technique. Most graphical representations of numbers are based on showing a part of a graphical object such as a bar or circle proportional to the value to be displayed or positioning or sizing an object according to the value. Colour can also be used; the simplest mapping from value to colour is to make the luminance proportional to the value (e.g. 0 = black, 1 = white). It is quite hard to determine a value accurately from a colour. This representation is only suited for detecting large errors in values. Figure 4.1 contains some examples of graphical representations of the real value 0.75 (assuming a minimum value of 0 and a maximum of 1). Characters and strings are, however, very hard to represent in a meaningful visual way as anything else than text.

4.1.3 Evaluation

Primitives can easily and clearly be shown as text, while most graphical representations have limited use at best. This is summarised in Table 4.2.

4.2 Array representations

Arrays have a wide range of sizes. For instance, some arrays may be short lists of objects, while others can be massive tables of numeric data. In order to handle this wide range of possibilities, several different visualisations are needed.

	0	1	2	
0	1	2	3	
1	4	5	6	

1	2	3
4	5	6

Figure 4.2: The array $\{\{1, 2, 3\}, \{4, 5, 6\}\}$ as a table (with and without indices)

0	0	1	2	
	1	2	3	
1	null			
2	0	1		
	Hello	World		

Figure 4.3: The array $\{\{1, 2, 3\}, \text{null}, \{\text{Hello}, \text{World}\}\}$ as a table of tables

4.2.1 Arrays as lists

One of the most straightforward ways of representing an array is by expressing it as a list of successive elements. These elements are usually separated by commas. For clarity in cases where nested lists may occur, lists are often enclosed in some sort of brackets (usually curly brackets). This technique is used in most command line debuggers. For example, an array containing the items 123, 456 and 789 would be expressed as $\{123, 456, 789\}$.

4.2.2 Arrays as tables

Arrays can also be displayed as tables. This notation is especially natural for two-dimensional arrays (by default, DDD displays 2D arrays as tables [62]), although it can be used reasonably well with tables with more or less dimensions. With three or more dimensions, some thought must be given to how to group the dimensions intuitively. One approach (which is used by default in e.g. Matrix) is to arrange arrays contained in a horizontally arranged array vertically and vice versa. Figure 4.2 contains an example of showing an array as a table.

If an integer is known to be an index into an array, it can be shown as an arrow or marker next to the corresponding array element. This technique is used in e.g. Tarrainím [40].

In many programming languages, such as Java, arrays are a type of object and multidimensional arrays are implemented as arrays of array references. In Java, an `Object` array may contain any objects, including `null` references and any array (including itself) [21]. In this case, the references between arrays are similar to the references between objects, and can be visualised in a similar fashion (as described in Section 4.3). Without indices, the result for a two-dimensional table is similar to the example in Figure 4.2. However, laying out each array contained within an array separately may result in columns or rows not lining up properly. This means that indices must be shown separately for each of the arrays contained in another array instead of using a single set of indices. However, this layout has the advantage of adapting better to array elements of different sizes (as shown on screen). An example is shown in Figure 4.3.

4.2.3 Arrays as plots

Arrays of numeric data can be represented as plots of various types (plots are also known as graphs; I call them plots to avoid ambiguity). For example, DDD represents one-dimensional arrays as two-dimensional plots with the array index on one axis and the value on the other (see Figure 4.4 for an example) and two-dimensional arrays as three-dimensional plots with

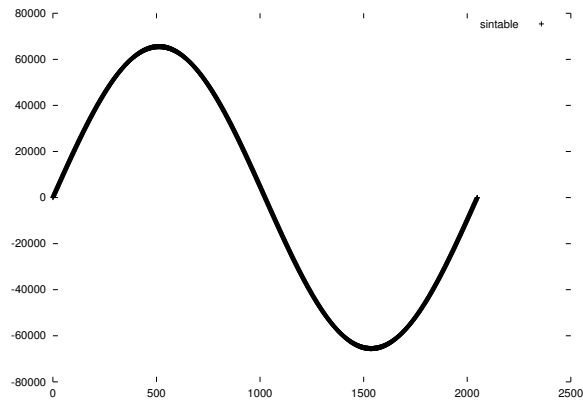


Figure 4.4: A DDD/gnuplot plot of a fixed point sine table (16 bit fraction, 2048 elements, one period)

a ———

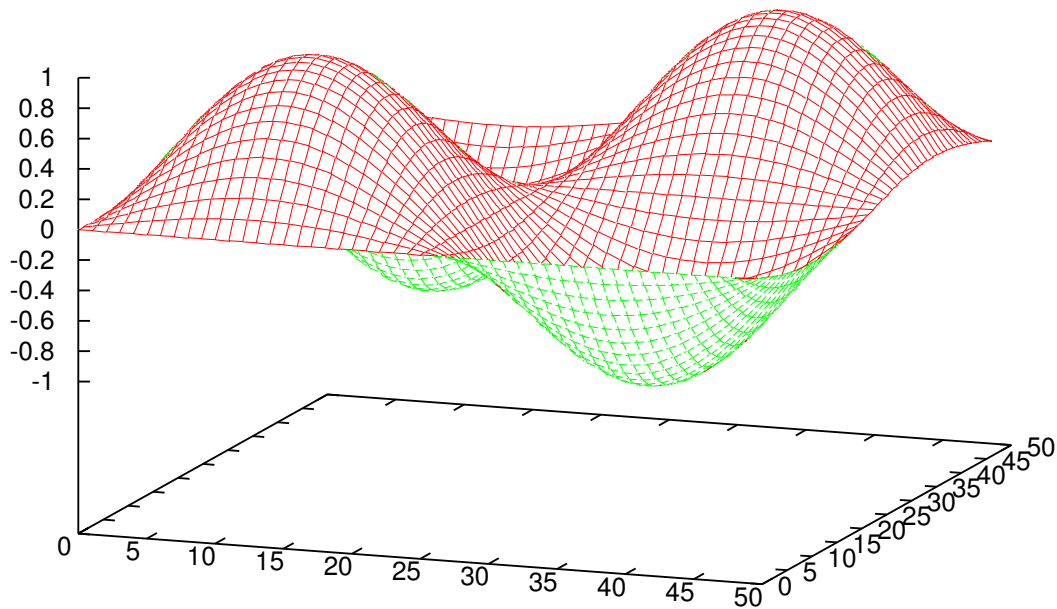


Figure 4.5: A DDD/gnuplot plot of $a_{ij} = \sin(\frac{i\pi}{25}) \sin(\frac{j\pi}{25})$ (50×50 elements)

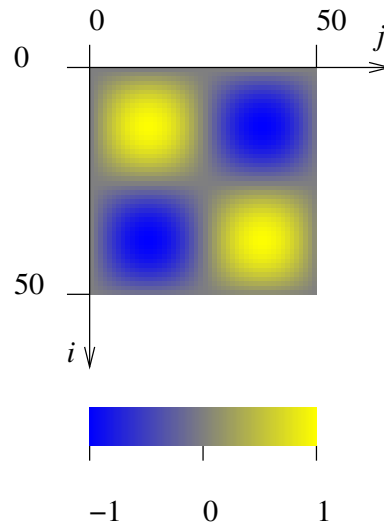


Figure 4.6: A colour plot of $a_{ij} = \sin(\frac{i\pi}{25}) \sin(\frac{j\pi}{25})$ (50×50 elements)

the array indices on two of the axes and the value on the third (see Figure 4.5 for an example) [62]. This technique is reasonably straightforward to implement and provides a good graphical representation for large arrays.

The plot drawing facilities can be generalised somewhat by allowing the user to select which axes to place values and indices on.

DDD handles compound variables (structs, objects, etc.) in plots by showing all numeric values contained in them [62]. This technique can be used to display arrays of objects containing multiple numeric fields.

4.2.4 Arrays as images

Two-dimensional tables of numerical values shown as colour values resemble images. In this case, it makes sense to draw each value as a pixel instead of a box; in other words, interpret the array data as a *pixmap*.

This representation makes particularly good sense in the case where the two array indices correspond to spatial dimensions in the context of the meaning of the data.

Images are slightly easier to comprehend than 3D plots, especially at small sizes. Colour can be used to help distinguish different values (e.g. by using a scale that goes from blue to yellow instead of black to white). Alternatively, many numeric values can be coded into the colour at once, although this easily becomes confusing. Small differences in values are harder to detect in an image than in a 3D plot. Figure 4.6 shows an array visualised as an image.

4.2.5 Evaluation

Lists are a straightforward way of expressing arrays, but tables are clearer in many cases. Plots and images can noticeably improve one's comprehension of large numerical arrays, but they are useless for most other types of array. This evaluation is summarised in Table 4.3.

Technique	Representation	
	Plottable arrays	Other arrays
Arrays as lists		
Arrays as tables		
Arrays as plots		-
Arrays as images		-

Table 4.3: Array representations

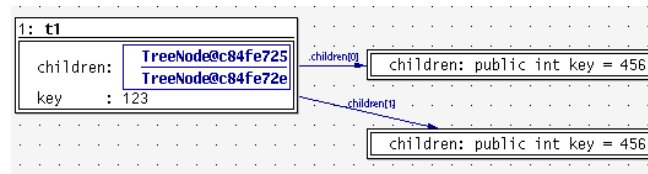


Figure 4.7: Example of objects in a Java program shown using DDD

4.3 Class and object representations

DDD [62] displays objects and structs as boxes containing the fields of the struct or class. References (pointers) to other objects are shown as arrows from one box to another (optionally labelled with the name of the reference field). Nested structs (as in C and C++ programs, for instance) are shown as nested boxes. An example data view from DDD is shown in Figure 4.7. Amethyst [39] also shows records and arrays as nested boxes. Null pointers can be shown as short arrows leading to nowhere or to a special symbol, as the absence of an arrow or as a special symbol or text string (e.g. “null”).

In Java, no object is stored inside another object; they are all separately allocated on the heap. This means that the program code does not distinguish between objects that are nested within another and those that are simply referred to. Thus, another criterion must be used to decide whether an object should be shown inside the object that refers to it or as a separate object.

4.3.1 Nesting or arrows?

The nested box representation makes sense in cases where an object unambiguously belongs to a single parent object (this corresponds roughly to composition in UML), while the graph representation makes more sense in the general case where an object may be referred to by several other objects. Deciding which one of these to use is not trivial, although some rules can be stated:

- Objects without references to other objects can always be shown nested inside the object that refers to them. This is quite intuitive for objects that wrap primitive values or contain only a few primitive values, especially if the objects are immutable, in which case there is little need to tell the difference between two objects containing the same data.
- Cyclic object nesting should be avoided. One way to avoid it is to use the graph representation. Another is to remove references that cause cycles (references from an object to one of the objects it is contained in) during abstraction, and put in a placeholder that identifies the referenced object. The former alternative is clearer in most cases. The latter is a convenient work-around in cases where nesting is desirable but causes cyclic nesting.

- Objects that take up a large amount of screen space should not be nested inside another object. Alternatively, maximum size restrictions can be imposed on objects and propagated to their subobjects as in Amethyst [39].

It should be noted that any object in memory in a C/C++ program can be referred to by a pointer, including the fields of a struct. In fact, in most implementations, every single byte of data and code in the program can be pointed to, irrespective of what it contains. This means that not only do the same problems with nesting apply to C/C++ programs; the problems are actually worse.

Ownership-based nesting

One way of determining whether objects unambiguously belong to a single object is to calculate the *ownership tree* of the graph (as defined in e.g. [45]) consisting of the objects and their references to each other. An object x owns another object y if all reference chains from the root object r (the only object that can be accessed without references from other objects) to y pass through x . An object x is an ancestor of y in the ownership tree if and only if x owns y . The parent of y in the tree is then the object that owns y but does not own any object that owns y .

The ownership tree suggests a natural nesting hierarchy with an interesting property:

Lemma 1 *An object y can only be referred to by objects that are owned by y 's parent p .*

Proof: p owns y but not x . Thus, a path exists from the root node to x that does not pass through p . Let us assume x refers to y . y can therefore be reached from the root node through x without passing through p , contradicting p 's ownership of y . Thus, x cannot refer to y . ■

If we nest objects inside the object that owns them, Lemma 1 implies that references to y can only come from p or an object z inside p . However, z is not necessarily contained directly in p (i.e. p need not be z 's parent). This means that references upward in the nesting hierarchy are still possible.

Ownership-based nesting has the advantage of automatically grouping objects together in a reasonably meaningful way. However, it has the distinct disadvantage that changes in the references between the objects can cause the nesting hierarchy to change radically, which may render it confusing to use.

As a Java method can read static variables and its own local variables at any time and start traversing the object graph from there, it seems reasonable that the root object should have references to all objects that are referred to by local and static variables. This, however, means that copying an object reference into a local variable or calling an instance method automatically means that the referenced object is considered to belong to the root. This leads to even more unintuitive changes in the nesting. Also, the ownership diagram depends on the currently active thread. In conclusion, ownership-based nesting can change suddenly in some quite common cases. It is therefore not suitable for visual testing.

Selection based on object type

Due to the problems with nesting, it seems best to default to arrows in all cases except objects that cannot contain references to other objects and other types for which the nested appearance is known to be desirable, such as arrays. It makes sense to allow the user to adapt the default settings to his own code and needs by specifying the types of object that should be shown as nested by default.

Manual selection

The user should also be able to override the automatic nesting choices (based, for example, on the class of the object) individually for every object. This is especially useful if an object that is usually shown nested is unexpectedly referred to by several different objects.

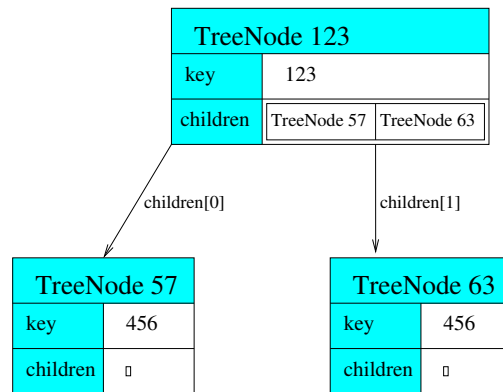


Figure 4.8: Example of labelled references

4.3.2 Static fields

In order to show the values of static fields, classes can be displayed in the same way as objects. In other words, classes can be considered singleton objects, where the static methods and fields are the methods and fields of the singleton object.

4.3.3 Labelling

To help the user identify objects and their types, each object should be labelled with its type and a unique identifier (e.g. “instance of `java.lang.String`, id 1234”, or more briefly “`java.lang.String` 1234”). Classes can be labelled with their name (e.g. “class `java.lang.String`”). The identifier for an object should be unique at least over the object’s lifetime. There is even less potential for confusion if the identifier is unique over the program’s entire lifetime.

In languages with pointer arithmetic, such as C++, pointers can be used as identifiers. Practically all debuggers for C and C++ use pointers to refer to heap-allocated objects.

In cases where the identity and type of an object are uninteresting, the label can be left out to save space. This is useful when visualising e.g. arrays (as illustrated in Figure 4.3) or strings.

4.3.4 Methods

Methods can also be shown in objects and classes for reference and to allow convenient method invocation. In most cases, however, displaying all available methods for every object is a waste of space. For invocation purposes, putting the methods for each object or class in a pop-up menu that is accessible e.g. by right-clicking on the object or class is almost as convenient as showing the methods all the time and occupies much less screen space.

Methods can also be shown as part of the code visualisation, as described in subsection 4.4.4.

4.3.5 Indicating the origin of a reference

When showing references as arrows, there are two different ways to indicate which field of an object contains the reference represented by an arrow.

One is to label the arrow with the name of the field and place the start of the arrow on the edge of the referring object, as in DDD. An example of this style is shown in Figure 4.8. I will refer to this style as *labelled references*.

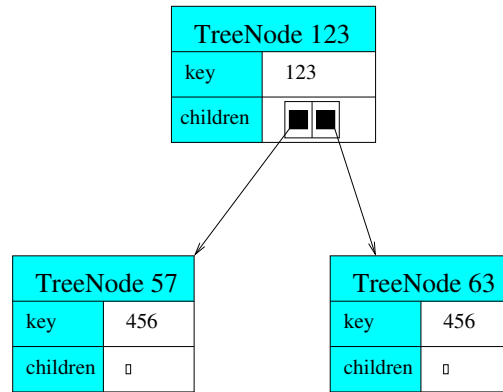


Figure 4.9: Example of nested references

Technique	Representation
	Objects
Objects without nesting	
Objects with nesting	
Labelled references	
Nested references	

Table 4.4: Object representations

The other is to place the start of the arrow in an area reserved for the field inside the object (the proposed pointer notation for Amethyst). Essentially, instead of a value the field of the object would contain a small placeholder from which an arrow extends. A null reference can be shown e.g. as a placeholder without an arrow. An example of this style is shown in Figure 4.9. I will refer to this style as *nested references*.

The latter approach seems to be less prone to clutter than the former, as the field names are shown tidily next to each other instead of appearing somewhere along the arrow. Therefore it is easier for the user to visually connect the arrow with the right field. Also, the latter approach works much better with objects nested within each other, as it allows arrows to clearly originate from a nested object instead of adding a complex label to describe the relation.

Also, the former approach becomes confusing if one wishes to mix nested and non-nested child elements in the the same array. This problem can be circumvented by displaying the labels of all the referenced objects inside the array (like DDD does) as shown in Figure 4.8 or showing indices for the objects that are shown nested inside the array.

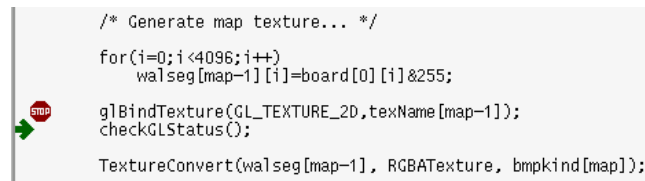
In short, the nested reference view is simpler, clearer and more flexible than the labelled reference view.

4.3.6 Evaluation

Showing a graph of objects without nesting is acceptable in many cases, but nesting can be used to improve clarity noticeably. Similarly, labelled references between objects are often acceptable, but nested references are clearer. This evaluation is summarised in Table 4.4.

4.4 Code visualisation

To make it easier for the user of a visual testing tool to understand the dynamic structures (data and execution) shown by the visual testing tool, the tool must show the link between



```

/* Generate map texture... */
for(i=0; i<4096; i++)
    walSeg[map-1][i]=board[0][i]&255;
glBindTexture(GL_TEXTURE_2D, texName[map-1]);
checkGLStatus();
TextureConvert(walSeg[map-1], RGBATexture, bmpkind[map]);

```

Figure 4.10: Part of a source code file as shown in DDD

these dynamic structures and the static structures of the program as described in the source code. In order to do this, the source code must also be visualised to some extent.

4.4.1 Displaying source code

The simplest way to visualise code is to display the source code text. Most visual debuggers (e.g. DDD [62, 63] and GVD [67]) just display the source code one file at a time, with annotations to indicate breakpoints, current execution position and such. Figure 4.10 shows part of a DDD code view.

4.4.2 Full-detail graphical representations of program code

Theoretically, graphical representations (such as parse trees, described in [4], for example) can be created for any programming languages without sacrificing any expressiveness, but these representations are not very useful in the usual case where the program has been written in textual form, as the textual form is more familiar to the programmer and usually more compact. Therefore, I will leave this approach out of consideration.

4.4.3 Colour pixel and line views of program code

Ball and Eick [7] have developed a wide range of code visualisations that show various properties of code as colourful figures. These representations are mostly designed to track changes between versions and display attributes such as code age or to provide overviews of various statistics about the code (e.g. nesting level).

These visualisations are intended for different problems than those addressed in this thesis, although some (e.g. viewing source code with colour highlighting and an extremely small font) may be useful as navigational aids.

4.4.4 Graphical hierarchies for classes

As the classes/interfaces and packages of a Java program (usually) correspond to files and directories in a file system (respectively), the tree browsing metaphors that are commonly used to browse file systems can also be applied to Java programs. The tree can be extended to the method level to allow convenient access to methods in large classes.

Practically all Java development and debugging tools use the package hierarchy (or the corresponding directory hierarchy) to browse source code. BlueJ uses the class inheritance hierarchy instead, displaying it in a form similar to a UML class diagram [27]. Both these views can be generated from the source code, from the compiled class files or from the classes loaded in the executing program.

Similar techniques can be used for C++ programs, although the correspondence between files and classes is not enforced by the language. Therefore, when examining C++ programs, the source code must be parsed to find classes. Also, C++ allows code outside classes, which complicates the issue. In this case, browsing the code using directories and files instead of packages and classes is easier to implement and possibly easier to use.

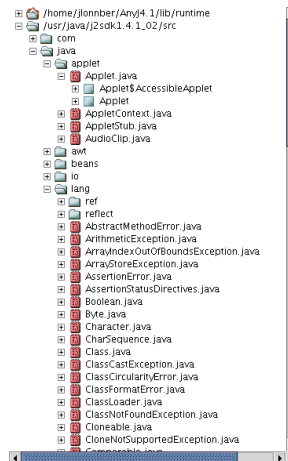


Figure 4.11: Part of a package tree of the standard library using AnyJ

Package tree

A package tree of a Java program can be generated by scanning the class path of the program. Using the source code requires additional parsing. Using the loaded classes makes it hard to access classes that have not been loaded. Therefore, the most satisfactory approach would be to base the tree on the contents of the directories in the class path. This will cause problems with custom class loaders, as there is no standard way to list classes available to a class loader. However, as long as only local files are used, directory listing can be used to look for available classes.

The package tree is used to select classes to view and edit in many IDEs. Figure 4.11 shows an example (taken from the AnyJ IDE).

Nested classes and interfaces in a Java program are stored as separate class files in the same directory as the class to which they belong [56]. Therefore, nested classes and interfaces will be shown as classes belonging to the package in which they reside, if the package tree is generated from directory listings of a file system instead of the source code.

The package tree can also be extended to contain the methods in each class. Many IDEs, such as AnyJ, do this to allow easier editing and viewing of source code.

Class diagrams

UML class diagrams can also contain information on other relationships between classes, such as composition, aggregation and references. It is hard to differentiate between these relationships in a Java program, but it is often easy to find these relationships by examining the types of the variables declared by each class. For example, BlueJ shows use relationships in its class diagram views. Showing references between elements in the program is further described in Subsection 4.4.5. Class diagrams can also show the fields and methods in each class and a lot of other information. See [20] for a more detailed description of UML class diagrams. Figure 4.12 contains a simple example class diagram.

4.4.5 Cross-references in code

Programs typically contain a lot of different classes and methods that refer to each other in different ways such as maintaining object references and calling methods. Getting a picture of how the parts of the program refer to each other should help one understand how the parts of the program interact and what the purpose of each part is. Tools such as Source-Navigator [77] can cross-reference a program and show references between elements in the

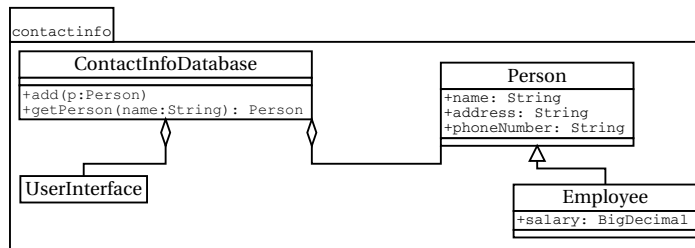


Figure 4.12: A class diagram of a simple contact information database

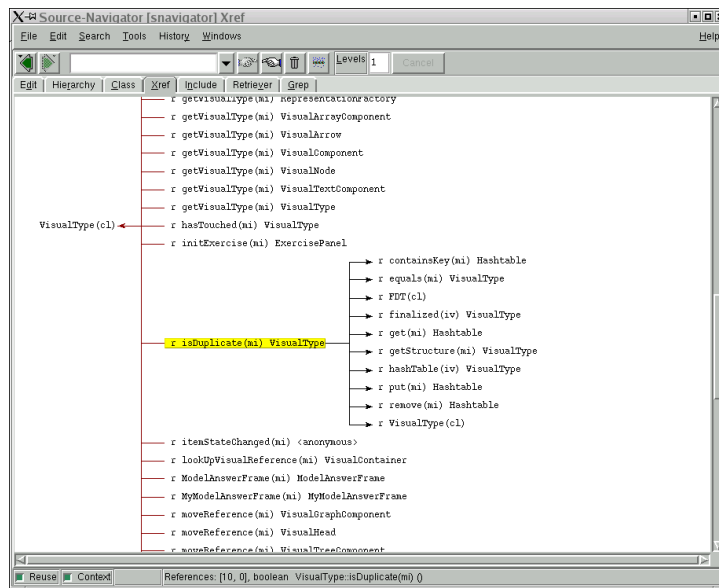


Figure 4.13: Browsing the reference graph of Matrix using Source-Navigator

program source code, such as all methods that refer to a class.

Cross-references can be visualised in a straightforward manner as graphs, but these graphs will be extremely complex if the entire program is shown down to individual method invocations and variable accesses. To avoid this problem, graphs can be drawn by starting from a specified element and allowing the user to request that elements referring to or referred to by an element should be added (Source-Navigator uses this technique). This allows the user to limit the graph to the references he is interested in, although the big picture is hard to get using this technique. A screen shot of Source-Navigator showing references to the class `VisualType` and from the method `VisualType.isDuplicate` in the Matrix source code is shown in Figure 4.13.

To get an overview of a large program, it may be more useful to show references between classes instead of delving into details on the level of individual methods and variables. This complements the detailed, user-controlled view used by e.g. Source-Navigator. References on the class level can be used as a part of UML class diagrams (see subsection 4.4.4).

4.4.6 Evaluation

Code visualisation can be split into two subproblems: a fine-grained view containing everything in the source code (such as the source code itself) and various coarse-grained views that provide an overview of the entire program with an emphasis on a particular aspect of

Technique	Representation			
	Package hier.	Inheritance hier.	References	Statements
Source code	I	I	I	III
Package tree	III	-	-	-
Class diagram	II	III	I	-
Reference graphs	I	-	III	I

Table 4.5: Code representations

the program, such as package trees, class diagrams and references graphs. Again, completeness means that everything in the code has a representation. Table 4.5 shows the evaluation of these code representation technique considered here.

4.5 Execution visualisation

The state of an executing program includes more information than the current data values and the source code. In particular, the current execution positions of each thread and the execution history are of interest to the programmer.

Instead of visualising the execution history of a program directly, information about causes and effects in the program can be visualised. The relationships between the causes and their effects are called the *causal relationships* of the program. The two most common forms of causal relationships in a program are between assignments and reads of the same variable and between a conditional statement and the statements caused to execute (or not) by this statement.

4.5.1 Tracing through source code

The most usual way of representing the execution of a program in visual debuggers is by displaying the source code with the current line highlighted or marked in some way. For example, in Figure 4.10 the current line is indicated using an arrow. This approach is very easy to implement and is familiar to most programmers, but requires a lot of screen space and does not integrate very well with a graphical data view.

The parameters with which the currently executing method was called can be added to the view (e.g. as an annotation to the formal parameters in the method header, although this may not be visible while tracing execution).

4.5.2 Highlighting objects

When a method is executed, this can be displayed by highlighting the object or class to which the method belongs in the data view and the running method. Similarly, when a field of a class or object is written or read, the field and the class or object to which it belongs can be highlighted.

4.5.3 Annotating objects

If the object view does not contain the methods, they must be added to the on-screen representation of the object in order to show that they are being executed. Multiple running instances of the same method can be shown by mentioning the method once for every invocation. Information on the location on the stack and parameters of each invocation can be shown as a part of this. The current execution position can also be shown for each active method. An example of this is shown in Figure 4.14.

This method can also be used to visualise the execution history of the program by annotating each object with the methods executed on it and the operations performed as

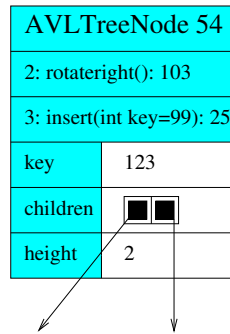


Figure 4.14: Example of an object annotated with executing methods (stack position, method name, parameters and current line number)

part of these methods. However, this will quickly become hopelessly cluttered. For clarity, the operations should be grouped by executing method and only the method invocations should be visible by default (see discussion of elision control in Subsection 5.4.1).

4.5.4 Drawing operations as connections

When an object calls another, they can be interconnected graphically (e.g. by an arrow) to show a method call. ProSasim uses this approach, which can be expressed conveniently using UML collaboration diagrams (see Subsection 4.5.8). This can be extended to manipulation of variables. Different representations could be used for read, write and call operations; various forms of arrows seem natural. Static method calls can be shown on the graphical representation of the class. Some of these ideas have been implemented in VisiVue.

4.5.5 Viewing the execution stack

Practically all modern programming languages have an execution stack or execution stacks of some form. The contents of an execution stack can be displayed in a number of ways. The most straightforward approach is to display the stack frames on top of each other as a vertical table. This suits the stack metaphor well. Each stack frame can then be considered a special form of object (the local variables in the frame can be considered fields) and displayed accordingly.

This allows the user to check which methods are active and the values of all their local variables and arguments. The current position in each method can also be added to the stack frame.

Most debuggers only show a list of calls and arguments in the execution stack view, and require the user to select a stack frame in order to view its local variables. This makes it hard to get an overview of the calls made by the program. Therefore, the local variables or at least the parameters of the method call should be shown in the stack frame.

4.5.6 Viewing the call tree

VCC [5] displays all function calls that have been made as a tree, with the currently active branch highlighted. By pointing to a node in this tree, the arguments and local variables in this function call can be seen.

Displaying the call tree for a thread is a generalisation of displaying the execution stack. It allows the user to examine the arguments of all calls that have been made instead of just the currently active calls. It is also a convenient way of structuring the list of operations performed by the thread.

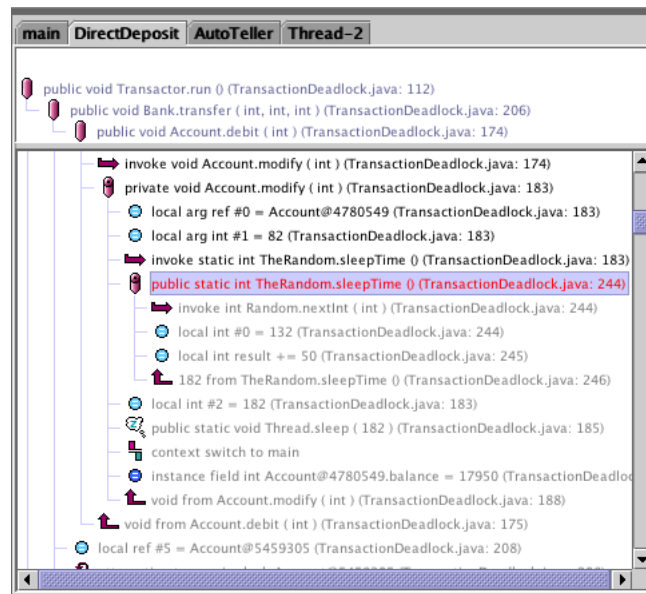


Figure 4.15: An example of RetroVue’s history view based on an extended call tree

To implement the call tree display, all method calls must be stored. Also, the values of local variables and arguments must be stored when a method exits. This can be done by copying stack frames whenever a method exits. Alternatively, if all local assignments are logged, the relevant information can be reconstructed from this.

RetroVue displays the entire execution of a thread, including all operations that change data values (e.g. assignments) as a tree structured by method calls [81]. This representation increases the information content of the call tree significantly. An example of a tree from RetroVue is shown in Figure 4.15.

The biggest problem with the call tree view is that it integrates badly with the data view, which makes it hard to see the connection between the objects shown in the data view and the operations in the call tree.

4.5.7 Sequence diagrams

UML sequence diagrams (described in e.g. [20]) show the flow of control and interaction between objects in an object-oriented program. In these diagrams, time is represented by the Y axis. They show objects as vertical lines with a box identifying the object at the top and boxes along the lines indicating execution and arrows between objects indicating messages between objects.

Only calls are usually considered messages. However, all operations can be shown as messages, at the expense of adding large numbers of messages from objects to themselves.

Sequence diagrams can be used to visualise the execution flow of a running program. As a sequence diagram of the entire execution of a program is going to be unmanageably large in most realistic cases, the user must specify the part of the execution of the program to view. One way to do this is by selecting a branch from the call tree. The sequence diagram can then easily be generated from the information in the call tree. Sequence diagrams can be used to display multiple threads that communicate using asynchronous calls [34]. An example of a sequence diagram is in Figure 4.16.

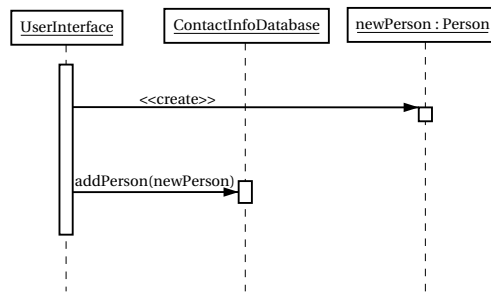


Figure 4.16: A sequence diagram of an insertion in the database shown in Figure 4.12

4.5.8 Collaboration diagrams

UML collaboration diagrams (in e.g. [20]) are another way to show interaction between objects. These diagrams show objects as boxes and messages (e.g. method calls) between them as arrows. The order and type of the messages is specified by labelling the arrows. The message type and parameters are shown as labels on the arrows.

The arrow labels include a sequence number for the message, the message (including its parameters) and (where applicable) a list of preceding messages in other threads [41]. Thus, collaboration diagrams can be used to show the operation of multithreaded programs, including the interaction between threads.

A collaboration diagram is more suitable for describing interactions between several objects than a sequence diagram. On the other hand, the order in which things happen is much clearer in a sequence diagram. Moreover, collaboration diagrams occupy less screen space than sequence diagrams.

Like sequence diagrams, collaboration diagrams are easily constructed from branches of a call tree. If the currently executing branch of the program is shown as a collaboration diagram, the resulting dynamic collaboration diagram is similar to drawing arrows for method calls (as described in subsection 4.5.4), but with numbering and without removing the arrows after the call is completed.

By displaying all operations as messages, collaboration diagrams can be extended to show all operations that occur, although this makes the diagram extremely cluttered.

4.5.9 Hybrid diagrams

The message passing arrows of a collaboration diagram can also be used to display calls and interactions between objects in a data view that shows the relevant objects, which eliminates the need for a separate collaboration diagram. Annotations can be used to display operations, statements or source code lines executed by an object and the current execution position in running methods. The resulting hybrid diagram describes both the state of the program and its execution history, but it may easily become cluttered and it is not in the UML.

As with annotation of objects, the operations performed by each invoked method must be elided by default to keep the amount of information shown manageable (see Subsection 5.4.1 for details).

An example of a hybrid diagram (depicting an AVL tree insertion interrupted just before rotating, with all executed instructions elided except for one of the currently executing insert method invocation) is shown in Figure 4.17.

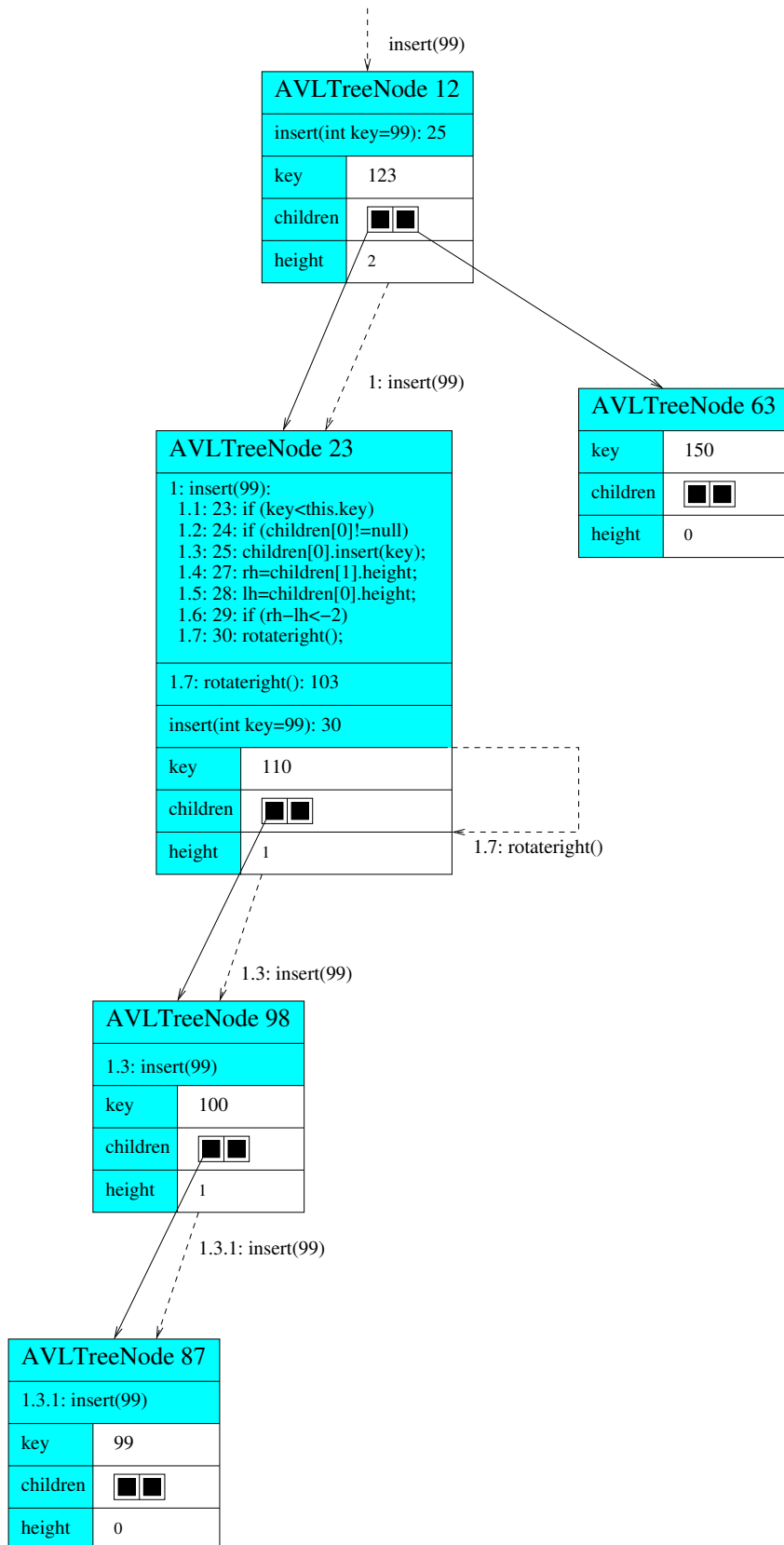


Figure 4.17: An example of a hybrid diagram

4.5.10 Assignment search

A simple way to allow the user to find the reason for the value of a variable is to search backwards through the execution history of the program for the last assignment to a variable. The result of this search is the statement that gave the variable the value it has at the current point of execution.

By searching through the execution history instead of attempting static analysis, side effects of e.g. method calls are automatically taken into account.

If the program is multithreaded in such a way that multiple operations can occur simultaneously on the same variable, it may be impossible to define a total ordering of read and writes on a variable. If variable reads and writes are atomic, this problem does not arise. In Java, read and write operations on `double` and `long` variables may be non-atomic [36]. This means that assignment search in Java programs may fail on `double` and `long` variables unless additional locking is introduced.

4.5.11 Slicing and dependence graphs

When looking for reasons for a program's misbehaviour, many programmers look at the (incorrect) value of a variable and try to find a reason for this value by tracing the execution of the program backwards. In essence, they try to concentrate on the part of the program that could have affected the variable value. Parts of programs that could have affected the value of a specified variable at a specified statement or point in the execution of a program are called *slices* and the process of calculating slices is called *slicing*. [60]

Definitions

Slicing uses techniques from data and control flow analysis to determine the statements that could have affected the value of a variable at a specified statement or point in the execution of the program. This is either done by static analysis, in which case the slice (a *static slice*) contains all the statements that can affect the value of the variable at the specified statement, or by dynamic analysis based on a specified execution history, in which case the slice (a *dynamic slice*) contains all the statements in the execution history that could have affected the value of the variable at the end of the execution history. In other words, the static slice is based on any possible execution history, while the dynamic slice is based on a specific execution history.

Slices are defined using *dependence graphs*. A static slice is based on a *program dependence graph (PDG)*, while a dynamic slice is based on a *dynamic dependence graph (DDG)*.

Definition 1 Program dependence graph (PDG):

A PDG is a directed graph. The vertices of a PDG are the statements in a program. An edge exists in a PDG from x to y iff $x \neq y$ and one of the following is true:

- x has a (*static*) *data dependency* on y , i.e. y writes to a variable, memory location or such that may be read by x .
- x has a (*static*) *control dependency* on y , i.e. x executes or does not execute depending on the path taken at a conditional branch at y .

Definition 2 Dynamic dependence graph (DDG):

A DDG is a directed graph. The vertices of a DDG are executed statements (one vertex for every time a statement is executed), and an edge exists from x to y iff $x \neq y$ and one of the following is true:

- x has a (*dynamic*) *data dependency* on y , i.e. x read a value written by y .

- x has a (*dynamic*) *control dependency* on y , i.e. y is the conditional branch that branched in such a way that x was executed and another choice at y would have allowed x to possibly not execute.

Definition 3 *Static slice:*

A static slice at statement e with respect to variable v is the set of statements reachable along the PDG from any statement s that can assign a value to v such that s is reachable from e without passing through any statement that assigns a different value to v [1, 2].

Definition 4 *Dynamic slice:*

A dynamic slice at the end of an execution history E with respect to variable v is the set of statements reachable along the DDG from the last statement in E to assign a value to v [1, 2].

In order for these definitions to work in multithreaded programs, the order of read and write operations must be defined for each variable; no operations of this type may occur simultaneously. As noted in Subsection 4.5.10, this is not necessarily true in Java without additional locking. In most programming languages, including Java, control flow in a thread can only be affected by conditional statements in that specific thread, which means that multithreading causes no additional problems for control dependencies.

Applications

Slices can be used to help a programmer locate parts of a program that may contain a fault, by allowing the programmer to select a variable in a statement and see the slice of the program with respect to the variable at this statement (e.g. by highlighting the slice in the source code display). This is used in Spyder [1] and Bandera [18].

Dynamic slices are (usually) smaller than static slices, and are therefore easier to work with when debugging [59].

By adding a few assignment statements to model method calls, object creation and other operations, DDGs can be defined for execution histories of object-oriented programs [64].

Slices often consist of a few statements scattered here and there in a program [60]. Displaying these statements by highlighting them in the source code is not very convenient, as the user must still look for the relevant statements in the source code. Furthermore, the slice contains only a small part of the information in the PDG or DDG. The highlighted statements correspond to the vertices of the graph, but the edges are not shown in any way. In other words, most of the information on dependencies between statements is lost when the dependency graph is converted into a slice.

The PDG may contain a vertex for every statement in a program, and is therefore probably too large to show completely. The DDG is in most cases even larger, as it can contain any number of vertices for each statement in the program; its size is only limited by the length of the execution history. However, the DDG provides information that is specific to a particular execution history, so it reflects the real behaviour of the program more closely than a PDG and thus suits visual testing better.

In most cases, the visualisation of a DDG must be limited to only a small part of the graph. The definition of a dynamic slice suggests that a good starting point is the last statement in an execution history to assign a value to a specified variable. Initially, the visualisation of the DDG only displays the vertex corresponding to this statement. The user can then select vertices and request that all edges from this vertex and the vertices at the ends of these edges be shown. This essentially means that the user can choose an incorrect variable value and then backtrack through the statements that could have affected the value to find the cause of the error.

Graphical layout

Definition 2 implies that if an edge exists from x to y , y was executed before x . Obviously, this precludes the existence of cycles in the DDG (an event cannot precede itself). Therefore, by placing statements in chronological order from top to bottom¹, a DDG can be laid out in such a way that if x depends on y , x is always below y . This makes the order of execution much clearer and makes arrowheads on the edges redundant (although including them may improve clarity).

In multithreaded programs, chronological order may only be a partial order (i.e. some statements have executed simultaneously so that none precedes any of the others). Assuming the assumptions made when defining a DDG hold, this poses no problem; any total ordering consistent with the chronological partial ordering can be used. Alternatively, statements that executed simultaneously can be shown at the same vertical position.

Application to Java programs

Constructing a DDG for a Java program appears to require modifying the JVM to keep track of dynamic dependencies. Also, displaying the execution of a Java program as a sequence of bytecode operations is not very useful, so the compiler should be extended to produce a mapping from bytecode to source code statements [59]. As a substitute for statements, source code lines can be used, although this will decrease the detail level of the view if several statements are written on a single line. Also, the bytecode instructions are at a very low level, so combining several bytecode operations into a single vertex in the DDG should improve the readability of the resulting graph. This can be done e.g. by treating a statement or a line as a single operation. In some cases, however, the option to break a complex expression evaluation into simple parts may be useful. Therefore, it is probably a good idea to allow the user to choose between using bytecode operations, statements and lines as the vertices of the DDG.

Most Java bytecode operations manipulate an operand stack that is specific to each execution stack frame. If bytecode operations can be observed individually, the locations on the operand stack can be treated as variables in the DDG.

Control dependencies can be calculated either by analysing the source code or the bytecode. When analysing the source code, the dependencies are more or less determined by the block structure of the source code. When analysing the bytecode, an algorithm that computes control dependency graphs for any control flow, such as the algorithm described in [19] and [4], must be used. However, analysing the bytecode has the distinct advantage that the bytecode has only a few simple operations that affect control flow, while the source code is more complex and may be optimised by the compiler.

The control flow in Java programs is complicated by two additional issues: exceptions and concurrency.

Exceptions make the control flow more complex, as exceptions can be caught in any of the calling methods and they can cause methods to terminate prematurely. Control flow graphs can be extended to include exception handling [51]. However, this extension makes the control flow graphs larger and harder to generate. For the purposes of visualisation it may be a better idea to ignore exceptions until one is actually thrown rather than clutter the dependency graph by adding the possibility of an exception being thrown to every time a field or method in an object is used; after all, a `NullPointerException` could be generated by any one of these operations. Similarly, every single read or write of an array element can cause an `ArrayIndexOutOfBoundsException` and every integer division or remainder can cause an `ArithmeticException` unless the indices or divisors can be shown to be always valid (which is hard to do in most cases). To ignore exceptions until they are actually thrown, one need merely ignore them when calculating control dependencies except for making the execution of the catch clause that catches the exception control dependent on

¹Obviously, this layout can be mirrored, rotated and otherwise transformed in a number of ways.

the operation that threw the exception. Obviously, adding exceptions in this way only works with dynamic analysis.

Concurrency adds synchronisation and communication dependencies between threads. Communication dependencies are already handled in the dynamic analysis case by the data dependency analysis, and synchronisation can be modelled by making operations that block dependent on the operations that caused them to be blocked and unblocked.

Algorithm

The processing needed to produce a DDG can be divided into two parts:

- Calculating the control dependency graph for the program.
- Collecting the information needed for the DDG while the program executes.
- Generating a DDG from the collected information.

The control dependency graph can be calculated using algorithms familiar from compiler technology. A good description can be found in [4].

While the program is being executed, the following information must be collected for each (bytecode) operation o that is executed:

- All reads by the operation (R_o).
- All writes written by the operation (W_o).

Additional information about the executed operations can be collected while they are executed to provide better visualisations, such as the bytecode or source code position of each operation.

Every read or write is a pair $\langle v, l \rangle$, where v is the value and l the location (operand stack element, array element, local variable or field) the value was read from or written to.

Assuming a sequence of operations o_1, o_2, \dots, o_n is being executed, the data dependencies between these operations can be calculated as the operations execute. A read $\langle v, l \rangle$ at operation o_j reads the last value written to l . In other words, the value read by o_j from l is data dependent on operation $o_{L(l,j)}$, where:

$$L(l, j) = \max_{(k < j) \wedge (\exists v | \langle v, l \rangle \in W_{o_k})} k$$

Obviously, if o_j writes to l , $L(l, j + 1) = j$. If not, $L(l, j + 1) = L(l, j)$.

Thus, instruction o_j is data dependent on:

$$D_{o_j} = \bigcup_{\exists v | \langle v, l \rangle \in R_{o_j}} o_{L(l,j)}$$

When drawing the DDG we also want to label the data dependencies with the value and location corresponding to the dependency, so we instead want to calculate:

$$D'_{o_j} = \bigcup_{\exists v | \langle v, l \rangle \in R_{o_j}} \langle v, l, o_{L(l,j)} \rangle$$

Therefore, the execution of the program looks something like:

$i \leftarrow 1$

while program still running:

Execute instruction o_i (storing R_{o_i} and W_{o_i}).

$D'_{o_i} \leftarrow \bigcup_{\langle v, l \rangle \in R_{o_i}} \langle v, l, L_l \rangle$

forall $\langle v, l \rangle \in W_{o_i}$:

$L_l \leftarrow o_i$

$i \leftarrow i + 1$

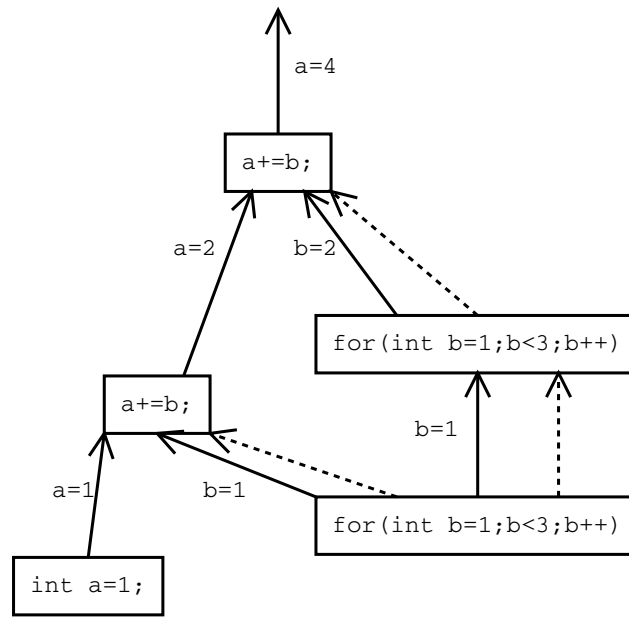


Figure 4.19: A line-level DDG

Drawing the DDG for a variable a after execution is then a simple matter of drawing a graph containing the vertices reachable along the edges defined in D' and C from o_{L_a} . Different types of edges should be used for data and control dependencies. For example, one can use solid lines for data dependencies and dashed lines for control dependencies.

Example

As a simple example of visualising a DDG with a Java program, consider the value of a at the end of the program:

```
public class ArithmeticTest {
    public static void main(String[] args) {
        int a=1;

        for(int b=1;b<3;b++)
            a+=b;

        System.out.println(a);
    }
}
```

By executing the program and calculating the DDG using bytecode operations as vertices, the graph in Figure 4.18 is produced. Control dependencies are shown as dotted lines, while data dependencies are shown as solid lines labelled with the corresponding value and variable (where applicable). This graph is hard to read for several reasons:

- Many of the data dependencies represent values on the operand stack, which exists only in the JVM, not in the Java language itself. This is reflected in the graph as data dependencies with no corresponding variable.
- There are lots of unnecessary vertices in the graph that do nothing but move data between the stack and local variables.

Technique	Representation		Completeness	
	Current op	Active calls	Current op	Active calls
Tracing in source code				
Highlighting objects				
Annotating objects				
Operations as arrows				
Execution stack				
Call tree				
Sequence diagrams				
Collaboration diagrams				
Hybrid diagrams				
Dynamic dependence graphs				

Table 4.6: Execution representations for current state

- It forces the user to think at the bytecode operation level.

Calculating the DDG using source code lines as vertices produces the graph in Figure 4.19. Grouping the executed operations by source code line removes most of the stack operations. However, if a statement is divided into several lines, intermediate values on the stack may become visible in the line-level DDG.

4.5.12 Evaluation

There are several contexts in which one may want to study the execution flow of a program in different ways. When one is watching the program execute step by step, a display of the current state of the execution of the program (the currently executing operation and a list of active method calls) is usually enough to allow one to understand the correspondence between the program's behaviour, its data structures and the statements that are executed. When trying out or testing a program, one may want to examine the sequence of method calls that has been executed. These two usage types have been evaluated separately in Tables 4.6 and 4.7. The current execution state consists of two data: the current position in the code and the set of active method calls. The execution history can be divided into method calls and other operations such as variable accesses, conditionals and loops. For each type of information, its representation (how easy the information is to extract from this notation) and completeness (how much of the information can be expressed using this notation) has been evaluated. For the purposes of this comparison, sequence diagrams, collaboration diagrams, annotations, hybrid diagrams and call trees are assumed to include all operations, not just method calls.

4.6 Summary

The representation effectiveness of the data visualisation techniques were shown in Tables 4.2, 4.3 and 4.4. Based on these tables, a visual testing tool should show primitives textually, arrays as tables, objects with nesting (where applicable) and with nested references. To further improve the visualisation of numeric arrays that can be meaningfully plotted, the option to view arrays as plots and images can be added.

As Table 4.5 clearly shows, showing the source code is a complete visualisation of the information in the code, but it is not a very useful representation in many cases. Package tree, class diagram and reference graph visualisations can be added to make it easier for the user to understand the package hierarchy, inheritance hierarchy and references in the code respectively.

Technique	Representation		Completeness		Causal
	Calls	Other ops	Calls	Other ops	
Tracing in source code	I	I	I	I	I
Highlighting objects	II	II	I	I	I
Annotating objects	I	II	III	III	II
Operations as arrows	II	II	I	I	I
Execution stack	II	II	I	I	I
Call tree	III	II	III	III	II
Sequence diagrams	II	II	III	III	II
Collaboration diagrams	II	II	III	III	II
Hybrid diagrams	II	III	III	III	II
Dynamic dependence graphs	II	II	II	II	III

Table 4.7: Execution history representations

Tables 4.6 and 4.7 suggest that showing the execution stack, displaying the execution history as a call tree and hybrid diagrams, tracing execution in the source code and browsing the dynamic dependence graph is sufficient for satisfactory visualisation of execution. This means that additional visualisations such as viewing operations as arrows, collaboration diagrams, highlighting or annotating objects and sequence diagrams are not really necessary, although some special cases may exist in which they are more convenient.

Chapter 5

Elision and abstraction

Elision (hiding unwanted information) and *abstraction* (hiding nonessential aspects of information such as implementation details) are the two key techniques used to keep the amount of information in a program manageable.

In well-written object-oriented code, abstraction and encapsulation are used to ensure that the functionality of a module of the code can be used, without being aware of the implementation of the module, through an interface that captures the essentials of the functionality of the module [26]. The goal of data abstraction in visual testing is to ensure that data structures can be viewed in a fashion consistent with their interfaces instead of their implementations in cases where the implementation is irrelevant.

The data, code and execution history views should all work with large amounts of information, and it therefore becomes necessary to provide convenient ways for the user to choose which information is shown. In other words, the data, code and execution history views need elision control.

Abstraction and elision techniques are evaluated using the point system described in Table 5.1.

5.1 Data abstraction

Abstraction of implementation details is necessary to keep a data view of a nontrivial program from becoming a bewildering mess of objects and references. Generally speaking, the user should have control over the degree of abstraction in order to help him concentrate on the parts of the program that he is working on instead of the internals of libraries.

The abstract data structures must be capable of referring to other data structures and other data structures must be able to refer to both the abstract data structures and other data structures inside them. In order to allow this, the abstraction must not break apart objects or classes into separate parts, as this makes it hard to refer to the object from elsewhere. As most abstract data structures will probably be implemented as objects or groups of objects of the same or similar classes, it makes sense to have the data abstracter receive an object

Score	Meaning
-	The technique does not do anything to satisfy this criterion.
I	The technique does not satisfy the criterion on its own, but does something that is useful with regard to the criterion.
II	The technique satisfies the criterion reasonably well in many cases.
III	The technique satisfies the criterion very well.

Table 5.1: Scoring system for evaluation of elision and abstraction techniques

and work out a representation for it.

In visual testing, producing an abstract data view consists of two tasks:

- Identifying the data structures that should be abstracted and the abstraction to use for each data structure.
- Converting the data to the abstracted form.

5.1.1 Identification by class or interface

In languages with an extensive standard library, such as Java, a large amount of useful data structures and interfaces are readily available to the programmer. Because of this, many Java programs use the standard data structures to provide convenient implementations of abstract data types. Recognising the classes that implement these data structures, which are part of the Java API standard [57], allows a large amount of data structure implementation details to be abstracted away. In this fashion, several data structures that can reasonably be displayed as arrays or lists, e.g. `java.util.HashSets` and `java.util.LinkedList`s, can be shown as arrays or lists instead of displaying their underlying implementation as debuggers usually do.

Essentially, the user has, in writing the program, explicitly marked several common ADTs (abstract data types) and FDTs (fundamental data types) as such. Not making use of this added information would be wasteful.

Although program visualisers usually use type definitions to determine the appearance of objects, most of them do not make use of prior knowledge of standard library routines. Instead, many of them (e.g. Eliot [58] and the Korsh-LaFollette-Sangwan system [32, 33, 50]) require the inclusion of special type definitions in the user program. However, Tarraingím uses the interfaces of objects to monitor changes to the objects and examine the objects [40].

Naturally, this technique can be generalised to data structures created by users, although this shifts the responsibility for adding support for the data structures to the visual testing tool onto the user.

Collections

Any object that implements the `java.util.Collection` interface can be considered a group of objects [57]. This group of objects can be shown as a list or a table. The implementation details of library classes that implement `java.util.Collection` are usually irrelevant from the programmer's point of view, and should be hidden by default. To provide additional functionality, some of the implementations and subinterfaces of `Collection`, such as `Set` and `List`, can have separate abstractions adapted to their specific properties.

Maps

The `java.util.Map` interface lends itself to abstraction in a similar way to `Collection`. The simplest visualisation is as a list of pairs, in which each pair consists of a key and a value (an object reference which can be displayed as a reference or a nested object, as usual).

Wrapper classes for primitives

The wrapper classes for primitives in `java.lang` can be shown as the primitives they contain.

Plots

The plots discussed in section 4.2.3 can also be considered abstractions of arrays.

Graphics

Graphical objects (such as `java.awt.Images` or objects that have a `paint` method that draws the object onto a specified `java.awt.Graphics` object) can be displayed graphically, instead of as objects. The `paint` method is used in subclasses of `java.awt.Component` to define the appearance of components, so the easiest way to detect this is to look for subclasses of `java.awt.Component`.

Many of these objects will probably be quite large, and in many programs most of them will probably be user interface objects with unchanging appearance. Cluttering the view with these by default is probably not a good idea.

Sound

Sound files, such as `javax.sound.sampled.DataLines` and `javax.sound.midi.Sequences`, can be shown as objects with “play” and “stop” buttons, which start and stop playback of the sound. This can even be generalised to video clips, although this is probably not very useful.

This addition is useless when debugging programs that do not process sound, so it is not very important.

5.1.2 Identification by patterns

UWPI [23] attempts to recognise ADTs by looking for recognised patterns of use; by examining how the CDT is used, it can determine what ADT it implements. While Java simplifies this process by encouraging the limitation of access to variables, it also provides predefined data types, either in the language or in the standard library, for many of the ADTs recognised by UWPI. Also, the type inference used by UWPI is inherently ambiguous in many cases. Furthermore, UWPI’s data type inferencer concentrates on recognising different uses of integers as e.g. Boolean variables or pointers, where a Java programmer would probably use `boolean` variables and references. UWPI is therefore better suited for languages with a limited set of primitives such as Pascal, C or C++.

This approach is not very useful on its own in Java, although it could be used as an adjunct to recognising known structures. For example, a tree can be constructed in Java by having a class of node objects, each of which contains a set of child elements. Here, determining that the nodes form a tree requires determining that each node contains a set of references to other nodes. To do this, we must first identify the set as such, and then check that it only contains references to other nodes. Finally, the graph expressed by the references must be shown to be connected and acyclic (if not, the resulting graph is not a tree). This poses a problem as the contents of the tree must be known in order to verify these properties of the references. Also, a structure that at one point appeared to be a tree may later be found to contain a cycle.

Because of the risk of incorrectly identifying structures by allowing differentiation based on properties that may change over time (for example, adding an edge may convert a tree into a graph with a cycle), it seems best to limit this abstraction mechanism to static analysis of the classes.

Linked lists

Linked lists have a simple structure: they consists of nodes that contain a link to another node (the next node) and some other data (usually a reference to a content object). The major problem with identifying linked lists based on their structure is that the references to the following objects may induce a cycle. Furthermore, several different objects may have the same next object, in which case the structure can be any graph where the out degree of every vertex is 1. Although identifying linked lists and displaying them as tables would provide a more compact representation for an easily implemented data structure, it is hard

to prove that the lists stay separated from each other. Alternatively, if a node is referred to from several different places, copies of the list can be shown in all of these places.

If a graph-like representation for references is used, linked lists will automatically be shown as chains of elements without any of the problems mentioned above. The visual layout may, however, not take into account that the linked nodes form a list.

In Java, it is arguably bad programming practice to implement linked lists that do not conform to the `java.util.List` interface, so there is little reason to attempt to detect linked lists based on their structure.

Trees and graphs

Trees and graphs are usually represented using nodes containing lists of children or neighbours (*adjacency lists*). If references between objects are drawn using arrows between boxes, graphs and trees of this type are automatically shown as graphs.

An adjacency matrix can easily be drawn as a graph; however, if one wishes to show the nodes as anything other than integer indices, a vertex data array must also be specified. Similarly, if adjacency lists are constructed using integer node indices, a mapping from indices to nodes must be specified. Specifying these mappings must in practice be done manually. For trees or graphs limited to a specific structure (e.g. binary trees), there may be several separate variables that form the adjacency lists. Otherwise, the situation is the same as for the general form. In conclusion, there seems to be little need to detect trees or graphs automatically.

5.1.3 Accessors and modifiers

In Java programming (and to some extent in other object-oriented languages), it is considered good programming style to provide access to data stored in objects through methods instead of allowing public access to fields. In fact, many books on object-oriented programming (such as [26]) and even the Java language specification [21] insist on this, as it allows much better abstraction and encapsulation than direct field access. One method (called an *accessor*) reads the value, and another (optional) method (a *modifier*) writes it. These are usually of the form¹ (for a value called `myValue` of type `Value`):

```
Value getMyValue();          /* An accessor. */
void setMyValue(Value v);    /* A modifier. */
```

Part of the time, accessors and modifiers are used to enforce access limitations on a field. For example, to allow any class to read the value, but only subclasses and classes in the same package to write to the value, one need merely make the variable storing the value `private`, the accessor `public` and the modifier `protected`. Modifiers are also used to check the validity of new data values before modifying object fields.

However, in some cases, accessors and modifiers may allow manipulation of a data value in a different form than the form in which it is stored. For example, the `toString` method in any Java object reads the data in the object and outputs it as a string. In these cases, it may be more convenient to access the data using these methods rather than directly access fields of the object. Essentially, these methods provide a way for the programmer to easily access a value that cannot be accessed directly by reading and writing a variable. Also, accessors and modifiers can be used to provide a common interface to data that can be stored in many different ways using different types of objects. By supporting this type of data access in a visual testing tool, we allow the user to examine his data in the same

¹Actually, the naming conventions for accessors in [21] differ depending on the intended meaning of the accessor. For example, an accessor that is supposed to convert the entire contents of an object into a value of type `Type` is called `toType`. These other forms of accessors can be identified using the same techniques as the simple case described here.

way that he manipulates it in the code he is working with by exploiting an abstraction that is already defined in the program code.

Accessors and modifiers can easily be identified if they are of the simplest possible form (where `field` is a field of `this`):

```
private Value field;

Value getValue() {
    return field;
}

void setValue(Value v) {
    field=v;
}
```

Recognising methods of this type provides no information except that it shows that a private field actually may be accessible outside the class in which it is defined. This information may be useful for elision purposes, as it provides an easy way to distinguish between private fields intended for internal state and private fields intended to be visible outside the package through an accessor. However, this information provides no additional abstraction.

If `field` can be a chain of fields (e.g. `obj1.obj2.fld`), a larger group of methods can be recognised, although changing one of the object references in this chain changes the field manipulated by the methods.

In the general case, the value manipulated by the accessor/modifier pair can be stored in a huge number of ways; it can be split, combined with other variables, compressed, encrypted, serialised, converted, and so on. However, the method pair still has the property that calling the accessor after calling the modifier returns the value passed as the parameter to the modifier. Identifying accessors and modifiers that do complicated processing on values and still have this property is a lot harder; for example, proving mathematical equations is actually a subproblem if arithmetic operations are allowed. Some simpler cases are analysable, such as those where data flow analysis allows one to prove that if `getValue()` is immediately preceded by `setValue(v)`, the value returned is always `v`. However, these cases only constitute a subset of the set of accessor and modifier pairs, and performing the data flow analysis is complex, although it is a well known problem in compiler design. Also, this technique cannot be used to identify accessors that do not have corresponding modifiers.

On the other hand, simply accepting every pair of methods with the right return and parameter types will probably yield a lot of false positives. The simplest method of filtering out false positives is to assume that the user's code follows the common naming convention of calling the modifier `set(value)` and the accessor `get(value)` for any name `value`.

Another criterion for an accessor is that it has no side effects; in other words, it does not change the state of any object. Specifically, the method and all the methods it calls contain no assignments to non-local variables. Checking this property requires parsing the potential accessor (either as source code or bytecode) and the code of all the methods it calls and ensuring that all variables that are set are local. This is quite a complicated solution, and it does not take into account the possibility of multiple counteracting side effects. However, when extracting data values to show in a debugger, a lack of side effects is a very useful property, as the debugger may execute the accessor at any time. Obviously, executing methods with side effects in a nondeterministic fashion (from the debuggee's point of view) is a good way to introduce new problems.

The ideas presented here for accessor/modifier pairs can also be applied to another common idiom: methods that add and remove items from collections and return the collection or an iterator for it. The problems with identifying these methods are more or less the same as for accessors and modifiers.

If read-only access is sufficient, the identification process can be limited to identifying accessors by looking for methods that return a value and have no arguments or side effects. This can be done automatically.

The values manipulated by accessors and modifiers can be considered equivalent to fields for the purposes of visualisation. As these values do not always represent the contents of real data fields, I will refer to them as *virtual fields*.

Executing accessors or modifiers on an object while it is being modified may cause invalid results, infinite loops, deadlocks or exceptions to occur in the accessor or modifier if the accessors and modifiers are not thread-safe. If they are thread-safe (and no thread-unsafe operations are being performed on the object), the accessor or modifier may only deadlock. Deadlocks and other aberrant behaviour can be avoided by running the accessor only when it is known to be safe to do so. For a properly designed object with accessors and modifiers, it is always safe to run an accessor or modifier when no other method is running on the object.

When using logging, virtual fields generated by accessors can only be shown reliably for historical data if the accessor was executed and its return value was stored at the time the data was current.

5.1.4 User-defined abstraction

In ambiguous cases (e.g. a tree implemented using a map from each node to its parent, an array that can either be viewed as a table or a graph or an accessor/modifier pair), cases where the actual implementation of a data structure is interesting, or cases where a new known structure is introduced, the user may want to specify a different representation than the default. A list of possible interpretations of a data structure (including the default general representation for unidentified objects) can be used for this.

The user may also wish to specify which fields of an object form the parts of the data structure (for example, in a map, the user may need to specify which field contains the keys and which contains the values). In order to do this, the user must also point out the variables that form the parts of the abstract data structure.

Many program visualisation systems rely heavily on manual identification and mapping of data structures to visualisation operations (e.g. Lens and Leonardo [16, 37, 38]). The mapping can be specified using a graphical user interface (as in Lens) or using a programming language (as in Leonardo).

Manual identification is very flexible, but it easily becomes time-consuming and error-prone if visualisations and abstractions cannot be reused easily. These problems are exacerbated if the visualisation is programmed as a part of this process. For example, Lens is not well suited for debugging for this reason [24].

Lifted fields

Instead of displaying the value of a field of an object, the user may wish to display the value of a field of an object that can be reached by following references in the object to be displayed. The user should be able to produce a chain of fields to traverse until the desired field is found; this field can then be shown in the object to be displayed as a virtual field. In other words, a field is lifted out from a referenced object, producing a *lifted field*.

5.1.5 Combining abstractions

When visualising complex data structures, it is important to be able to combine abstractions. This allows the user to easily create complex abstractions that would otherwise require additional code.

In order to do this, abstractions should be designed to read and write data in the same form as the visualiser. This means that any abstraction that can be visualised can also be used as part of a combined abstraction.

As an example consider showing the attributes of an XML element using the DOM (Document Object Model) API (a part of the standard Java API [57]). The attributes are returned as a map containing attribute objects with names and values. Using two accessor virtual fields, the names and values of the attributes can be extracted as fields. The map can be converted to a table of attributes by specifying the methods to get the table length and an item in the table. This table of pairs can then be interpreted as a map by specifying the fields of the pairs to use.

5.1.6 Data abstraction model

The abstraction techniques described in this section can be expressed using virtual fields and *convertors*. Convertors output objects that implement the interface of one data type based on the data in an object of another type. For example, the abstraction for `Collections` described earlier can be implemented as a convertor that outputs a table and uses the `Collection` interface to read and write elements.

Convertors can have settings that determine how they interpret the structure they convert from and properties of the structure they convert to. For example, a convertor that produces an array from a structure with methods to get and set an element has at least three settings: the two methods to use and the field to read the length from. A convertor applied to a specific type of data structure with specified settings can be considered an *interpretation* of the real data structures. By applying convertors only when necessary, a lot of storage and processing can be avoided.

All virtual fields can be rewritten as a field containing a suitable convertor. Therefore, only one single virtual field type is necessary: a converted field.

An interpretation defined for objects of a class is also valid (syntactically, at least) for objects of its subclasses. Similarly, an interpretation defined for an interface is valid for classes that implement it. This means that the interpretations defined for the ancestors of a class (the ancestors of a class or interface are its superclass, the interfaces it implements and their ancestors) can be used for the class. Similarly, if no default interpretation has been specified for a class, the default interpretation of any of its superclasses or interfaces can be used.

Deciding which ancestor's default interpretation should be used is not trivial. However, a few useful rules of thumb can be devised:

- If a class has its own specific interpretation, it is usually better suited for that class and its subclasses than any ancestor's view. Similarly, the interpretation of an interface is better for classes that implement the interface or a subinterface of the interface than the interpretation of a superinterface of the implemented interface.
- If a class is similar to the class for which a visualisation is defined, the visualisation will probably work reasonably well.

Thus, assuming we can quantify “similarity” (or conversely, “dissimilarity”) between two classes or a class and an interface, we have an ordering on the suitability of interpretations of ancestors, and can choose the most suitable interpretation. One possible definition of dissimilarity between a class and a superclass of it is the amount of overridden or added methods. The dissimilarity to an interface can be defined as the amount of added methods.

To allow the user to switch between different valid interpretations of an object easily, each class should have a set of defined interpretations. The user can then easily choose which one is used for visualisation. If these interpretations are identified by name, they can be used as input for convertors to combine abstractions without affecting the visualisation.

Technique	Abstraction	Auto view control	Manual view control
Identification by class/interface			
Identification by patterns			-
User-defined abstractions		-	

Table 5.2: Data type identification techniques

5.1.7 Evaluation

Producing a data representation with abstraction involves two tasks: identifying the data structures and converting them into a suitable form.

All the identification techniques examined here require manual definition of some form of rules for the recognition of implementations of a data structure as well as a suitable visualisation for the data structure. The different approaches only differ in how much of the work is done by the user (reflected in the automatic view control rating), how well the user can adapt the visualisation to his needs (manual view control) and how close the visualisation can be to the user's conception of the data structure (abstraction). Identification rules should recognise as many of the data structures to which a visualisation can be applied as possible.

Pattern-based recognition techniques require the definition of usage rules for the data structure. The rules for this type of identification can be hard to produce, and they can usually only detect a specific implementation of a data type. Due to these problems, their applicability is limited.

The best approach seems to be to extract as much identification information as possible from the program itself. For each class in an object-oriented program, the implemented interfaces and superclasses are specified. This, combined with a set of visualisations for known data structures that can easily be extended and adapted, should cover most cases.

Identifying by class or interface is automatic, reliable and desirable as long as a view where the known data structures have been abstracted is acceptable. In practice, some manual intervention may be needed to get the desired view. Table 5.2 summarises the evaluation of data abstraction techniques.

The data abstraction model described in 5.1.6 is primarily based on known data structures and manual identification, as these techniques complement each other and are simpler than identification by patterns.

A lot of different ways to extract information from a data structure once it has been identified exist. The suitability of each method depends on the data structure to which it is applied and the desired visualisation. Properly designed data structures in object-oriented programs can usually be manipulated through their methods.

5.2 Data elision

Elision control (allowing the user to decide what to leave out) is of great importance in the data view of a debugger or program visualiser. On the one hand, all variable values should be accessible. On the other hand, displaying the entire state of a program will in most cases cause the interesting information to be drowned out by irrelevant detail. As the tool cannot always guess correctly what information the user is interested in, manual elision control is necessary. However, in order for the user to be able to effectively use the tool, the need for manual elision control should be minimised. Thus, the tool must be able to make the most common elision control decisions automatically.

In the context of a visual testing system, the goals of data elision and data abstraction overlap somewhat. For example, implementation details of library routines can be hidden both by abstracting the implementation of the data structure as described in section 5.1 or

by simply hiding the library object or its contents. However, while data abstraction provides access to the information in a form that is easier to use, data elision simply hides it.

5.2.1 Automatic elision control

In most cases, the user is not interested in the implementation of library classes. Therefore, library classes and their instances should be shown at as high an abstraction layer as possible with all internal state hidden. Also, library classes and their instances should (under normal circumstances) only be shown when they are referred to by classes in the user program.

The detail level shown when an object is displayed should be limited to ensure that the graphical representation of the object is not unreasonably large. In particular, this means that large objects should not be displayed fully (DDD hides parts of objects that are too large to achieve this). Two common causes are objects with large amounts of fields and objects with large amounts of other objects nested within them. In both cases, hiding the contents of large objects by default helps a lot.

Some objects can vary a lot in size; these are arrays and strings. For strings, the simple rule of truncating the string at a specified length often suffices. For arrays, a more general idea is required. One approach is to define a maximum size for the graphical representation, and remove layers of detail until the graphical representation is small enough. In the case of nested arrays, this would mean that some or all of the levels of arrays would be elided. Another approach is to display the array elements by default only if they are sufficiently few.

5.2.2 Manual elision control

Visual debuggers, e.g. DDD and GVD, usually display data only if the user has explicitly specified that he wishes to see it. This choice is usually made one object at a time, with the ability to hide and reveal elements of nested data structures such as arrays. This is easy to implement and quite efficient, but is also quite tedious to use; in order to examine a large data structure, the user must spend a lot of time pointing out objects to show (or hiding irrelevant information). On the other hand, this provides very fine elision control.

Algorithm simulation tools, such as Matrix, by default display all the data structures that are being manipulated by the user at the FDT level. Many visualisers, such as Jeliot and JDSL, have very limited elision control, as they are primarily designed for educational use with simple examples [29].

For practical use, coarse-grained elision control must also be available. In most cases, the programmer is interested in testing or examining a specific part of a program. Instead of controlling the visibility of one object at a time, showing and hiding classes and all their instances at a time would be useful in these cases. Allowing the user to show or hide all of the instances of all classes in a package would allow the user to conveniently examine everything that happens in a specific package.

The manual elision control should always allow anything hidden by automatic elision control to be shown.

5.2.3 Evaluation

Data elision and abstraction overlap somewhat, as abstraction also elides implementation details. Automatic and manual elision, if implemented correctly, essentially increase the automatic and manual view control ratings (respectively) as shown in Table 5.2 by one point. This evaluation is based on the improved view control and flexibility provided by these techniques.

Technique	Causal understanding	Manual view control
Structure-based elision	-	
Slicing		

Table 5.3: Code elision control

5.3 Code elision

Obviously, showing all of the code in a program at once is unpractical in all but the most trivial cases. Therefore, the user must be allowed to choose which code is shown.

5.3.1 Structure-based elision

The most common way to filter source code views is to allow the user to select source code files to view. This can be done using whatever traditional file selection techniques (an “open file” dialogue box, for example) or using a package tree (see Subsection 4.4.4). This is easy to implement, and corresponds closely to the way in which most developers edit their code.

If a source code file contains a large amount of methods, it may be convenient to allow the user to choose a method to view. As noted in Subsection 4.4.4, the methods in each class can be added to the package tree. Alternatively, a list of methods from which the user can choose can be shown for each class (e.g. as a drop-down list box or pop-up menu).

The idea of selecting source code to view can be extended higher up in the code structure hierarchy by allowing the user to select code to view a package at a time. This is too coarse-grained to be useful.

More fine-grained alternatives are possible, such as collapsing blocks in the source code (in C, C++ and Java, blocks are delimited by curly brackets (`{ }`)). Implementing this requires at least partial parsing of the source code files.

5.3.2 Slicing

Static or dynamic slicing, as described in subsection 4.5.11, can also be used to elide parts of a program that could not have affected a specified variable value at a specified point in the program execution. Dynamic slicing can be used to find the parts of the program that could have affected the value that was actually stored in a variable at some specified point in the past, while static slicing can be used to examine all of the code that can possibly affect a variable’s value at a specified position in the program code.

5.3.3 Evaluation

The code elision techniques can be evaluated according to how well they support causal understanding of the program and the degree of manual control over the view they provide. This evaluation is shown in Table 5.3.

5.4 Execution elision

The full execution history of a program is in most cases far too large to visualise completely. Therefore, mechanisms to limit the view to interesting parts of the execution history are absolutely necessary.

5.4.1 Call tree-based elision

By choosing which branches to view in the call tree, the user can make it easier to concentrate on a part of the execution history. In multithreaded programs, this also limits the execution view to a single thread, as each thread has a separate call tree.

Choosing the outermost call

For visualisations that show the operations or calls performed by a called method, one obvious elision technique is to choose the method call to use as the outermost visible method call. In the call tree visualisation, this is the root of the tree. In the connected objects, collaboration diagram and sequence diagram visualisations, this is the first message. This elides all of the operations executed before this method invocation and after its return. This elision type is useful when the user wants to examine the operations executed during a specific method invocation.

In the execution stack view, a similar elision can be performed by removing all items below a certain point. This is not very useful in practice unless space is very limited.

Collapsing call contents

In execution history views, the operations performed by a method can be left out, meaning that the method invocation is represented as a single operation. In the call tree view, this means that a branch is collapsed to a single node. In the connected objects, collaboration diagram and sequence diagram visualisations, this removes the arrows representing operations inside the collapsed method invocation and objects that are no longer referred to by an operation.

A similar operation can be used on annotated objects and hybrid diagrams to keep the execution history view manageable. By default, the annotated objects or hybrid diagram should only show the operations performed by one method invocation, including the calls it made. The user could then expand any of the displayed method calls to show all operations executed by the corresponding method invocation.

This elision type is useful when the user knows what has happened during a method invocation or does not need to know. Library call contents should always be elided by default, as the user usually does not need to know what library code actually does.

5.4.2 Filtering

By leaving out certain types of operation from the execution history view, the user can concentrate on the remaining types of operations.

Filtering by operation type

The user is often only interested in a certain type of operation. For example, variable reads have no impact on the state of the program and are uninteresting when one is looking for the part of a program that sets a specific variable. It is therefore useful to be able to choose which operations are shown in a history view. Most method calls must be left in the call tree to allow operations performed by these methods to be shown in the right place in the tree.

Filtering by accessed element

When the user wants to find a reason for a value written to a variable or study how the value of variable changes during the execution of a program, it is useful to be able to limit the visible operations in a history view to operations that access a specified variable. In the

Technique	Causal understanding	Manual view control
Call tree-based elision	-	
Filtering	-	
Dynamic slicing		

Table 5.4: Execution history elision control

case of call trees, most of the method calls must remain in the tree to allow the variable accesses to be placed within the correct method invocation.

5.4.3 Dynamic slicing

Dynamic slicing, as described in subsection 4.5.11, can also be used to elide parts of an execution history that could not have affected a specified variable value at a specified point in the execution of the program.

5.4.4 Evaluation

The execution history elision techniques can also be evaluated according to how well they support causal understanding of the program and the degree of manual control over the view they provide. This evaluation is shown in Table 5.4.

5.5 Summary

Used on its own, abstraction does not provide good view control, unless data structures are manually identified and their visualisations manually specified (in which case the elision is performed as a part of the abstraction).

As can be seen in Subsection 5.2.3 and Tables 5.3 and 5.4, none of the elision control techniques on their own provides good causal understanding or view control. However, they can be used to enhance the effectiveness of the code and execution history representations in causal understanding. Also, the techniques complement each other well. The manual view control rating of a tool (as defined in Table 3.1) in which all elision control techniques are available and used together with manual abstraction control should be |||.

Chapter 6

Controlling the debuggee

Giving the user the ability to control the program that he is examining is an important part of visual testing. This corresponds to execution control and data modification as described in Section 2.1.

6.1 Data modification

Being able to change data values interactively in a running program allows the user to try out the effect of different data values on a part of a program without writing additional test code or invoking methods. This ability can also be used to construct inputs for method invocations.

6.1.1 Textual editing

One way to change the value of a variable is to select it (in the data view or by entering its name) and enter a new value. This, of course, only works well if the variable can be meaningfully and unambiguously expressed as a string. This technique is unpractical when working with object references, unless they can be expressed using other variable values or similar expressions. However, primitives can easily be edited textually using the representations described in Subsection 4.1.1.

If only a simple value can be entered, this is easy to implement but tedious to use. If more general expressions are allowed (including method calls; see 6.2.1), this technique is much harder to implement, but more flexible. Source code for this type of expression evaluator is widely available (e.g. in the JDB sources), but using it may cause licensing problems.

6.1.2 Graphical reference manipulation

One way to specify that one wishes to change a reference variable to point to another object is to drag the object onto the reference variable. Alternatively, if the reference is displayed as an arrow, the end of the arrow can be moved to point to the desired object. Both of these techniques appear in Matrix (for nested and referenced objects respectively).

6.1.3 Graphical primitive entry

Entering primitive values graphically is possible, but not very useful in the general case.

In cases where precision is not important and the range of the value is known, floating point values can be specified using a variety of sliders, knobs and other GUI elements. However, in most languages the range of a floating point variable is too large to allow

useful data entry in this fashion. Sketching curves using a mouse may be useful for entering information into floating point arrays, but only in a very limited set of circumstances.

Textual primitives such as characters and strings can in most cases only be meaningfully manipulated textually.

6.1.4 Graphical expression entry

Expressions can be entered graphically by constructing a tree consisting of operators and methods (non-leaf nodes) and values (leaf nodes). The children of each operator node are the arguments the operator is to be applied to. Operators can be selected from a predefined list. Objects, variables and methods can be specified by clicking them in the data view or dragging them from the data view. Primitive values can be copied from the data view (e.g. by dragging) or entered graphically or textually.

This is essentially the same as editing a parse tree for an expression graphically. In practice, this technique is much more time-consuming than typing the corresponding expression and less familiar to the programmer.

6.2 Method invocation

In order to invoke a method, one must select the method to run, the object to run it on (for non-static methods) and the arguments. These can be selected in several ways. For the purposes of a user interface, object creation can be viewed as invocation of a (static) constructor method.

6.2.1 Textual invocation

Textual method invocation can be handled essentially like editing of values (see section 6.1.1); the user enters an expression (which may include method calls) to be evaluated. The result of the expression is then printed textually or shown graphically. This is used by most debuggers, although it requires the ability to parse a significant part of the language to be useful.

6.2.2 Graphical invocation

The object or class to call a method on can easily be selected by pointing and clicking if methods are shown for objects or the methods are easily accessible through a pop-up menu. However, choosing the arguments for a method call is harder. In the case where a method requires no arguments, a single click is sufficient. In the case where a method requires one argument, this argument can be selected e.g. by dragging and dropping the object to be used as a argument onto the method to be run in the object on which the method should be run. E.g. Matrix uses this technique.

In the general case, the user must be allowed to select several arguments (both primitives and references). Primitives are easily specified using textual input, while references are more easily specified using drag and drop. For clarity, the primitives should be grouped together in a method call object that can either be nested in the box of the object containing the method or shown separately, as in BlueJ [27]

See also Subsection 6.1.4 for another way to invoke methods graphically.

6.3 Starting and stopping execution

An important part of debugging is being able to interrupt the execution of the debuggee and examine its state. Most current debuggers provide breakpoints, single stepping and watchpoints to address this need.

Breakpoints were introduced in FLIT [55] and have changed little since that time. A breakpoint is essentially a request that a program be stopped at a specified line or statement. In debuggers with graphical user interfaces, breakpoints are usually specified by pointing at a source line and requesting from a menu that a breakpoint be put there. In textual debuggers, the source code file and line number are usually specified.

Single stepping is a feature that allows the user to execute one line, statement or instruction at a time of a program. It is often used to examine the execution of a program in detail.

Watchpoints (also known as watches) are requests that a program be stopped whenever a variable is accessed or modified. They are usually specified by entering a variable name or pointing at a variable in the source code or data view.

When examining a multithreaded program, it is usually helpful to be able to be able to disable the execution of threads at will. This helps the user concentrate on the interesting parts of execution and aids in debugging race conditions.

6.4 Summary

Even in a graphical user interface, textual entry for expressions and primitives is a lot more practical than graphical entry. Editing references by dragging arrows is often more convenient than manipulating them indirectly through reference variables. It is easier to select objects by pointing and clicking than by referring to them textually using reference variables. A combination of textual and graphical input seems to be the best way to select values and construct expressions for data modification and method invocation purposes.

Breakpoints, watchpoints and single stepping are more or less taken for granted nowadays, and should therefore always be available in any tool intended for serious use.

Chapter 7

Implementation

Implementing a visual testing tool is by no means an easy task. This chapter describes problems related to implementing a visual testing tool and possible solutions to these problems.

7.1 Connection to debuggee

As the process of connecting a debugger to a running program is highly dependent on the language and the technique used to execute programs run in it, this section is limited to Java code running on a JVM (Java Virtual Machine).

7.1.1 Instrumentation of code before or at compilation

Many program visualisers, such as Eliot [58], Jeliot [22], VCC [5] and UWPI [23], automatically add visualisation code or calls to visualisation code to programs before or during compilation. In order to do this, a preprocessor must be written to examine the program and add the visualisation. While this can be done portably, using this approach requires that all code that one wishes to analyse be compiled by or preprocessed before compilation by the visualiser's preprocessor or compiler. This approach also requires extensive additions to all code that is run in order to support e.g. examining the execution stack (for example, every method call and return must be augmented to update a copy of the stack). Furthermore, instrumenting the code before compilation provides only limited control over execution order.

The advantage of instrumenting code at compilation is that it is easy to maintain a mapping between the compiled code and the source code. For example, slicing and dependence graphs work better with detailed information on which statement corresponds to which bytecode operation.

Adding operations to methods may change their execution times and result in different (but equally valid) execution orders in some parallel programs.

This approach requires full source code for all classes that are to be instrumented, including the standard library.

Special care must be taken when using instrumentation to ensure that variable writes cannot occur between a variable read or write and its corresponding instrumentation call. This is not a problem for local variables. However, fields and array elements can be accessed by many different threads at once. Adding locks as a part of the instrumentation is only a partial solution, as uninstrumented code may be able to access the variables. Without the additional locking, a multithreaded program may be able to interleave writes and reads in such a way that the debugger shows incorrect variable values.

7.1.2 Instrumentation of compiled code

Visualisation calls or other data collection mechanisms can also be added to code that has already been compiled by adding bytecode instructions to the method definitions in the class files. This can be done using a custom class loader that instruments the code when it is loaded, or by instrumenting the class files in advance. ODB can use either of these techniques. When a modified class loader is used, code that is loaded using a user-defined class loader will not be instrumented. [35]

By adding a debugger call to every bytecode operation in the debuggee, all state changes to the debuggee caused by bytecode can be tracked. In some cases, it may be useful to call the debugger both before and after the operation to find out both values read and values written by the operation. However, not all operations are bytecode; any non-trivial Java program must execute native code for e.g. I/O. This native code may read and modify the state of the objects and classes in the JVM using the Java Native Interface (JNI). Therefore, any system that relies solely on bytecode instrumentation to collect information on state changes will produce incorrect results if a native method modifies objects in the program. Similarly, in order to track execution of system code, the standard libraries must also be instrumented. Library classes that are not instrumented will cause similar problems to native code.

As with instrumentation of code at compilation, instrumentation after compilation may change the concurrent behaviour of the program. Similar care must also be taken to ensure that read and write operations maintain their order in the face of concurrency.

7.1.3 Instrumented interpreter

By running the user programs on an interpreter (such as a Java Virtual Machine), one can examine and modify every aspect of the running user program by instrumenting the interpreter. Leonardo, for example, implements this approach by running programs on an emulated machine.

In the case of Java, modifying the JVM to collect the necessary information seems easy at first. However, modifying the Java Virtual Machine causes a few problems:

- The solution is tightly bound to a specific version of a specific VM. This effectively means that the debugger must be rewritten at least partially every time the VM changes. It also complicates support for multiple platforms.
- Modularity suffers; most VMs are probably not designed for internal modifications and their internals are often not as well documented as the APIs they implement. This makes the process of editing the VM more error-prone than accessing it through a published API.

Several JVMs are available freely as source code. Adapting one of these should not be too hard. Unfortunately, no freely available JVM seems to exist that includes support for all the native methods in the standard Java library. For this reason, a lot of software is compatible only with Sun's JVM and library implementation, which limits the generality of a solution based on one of the free JVMs. Sun provides source code for their JVM, but the license is complex and has some interesting side effects, such as giving Sun the rights to do practically anything they like with everything you write that conforms to a part of the Java specification (including, arguably, work that is not based on their source code).

In theory, the Java language can be interpreted directly from source code, although the most common solution by far is to compile it to bytecode and interpret the bytecode instead.

7.1.4 JPDA

The *Java Platform Debugger Architecture* or *JPDA* [71] provides support for easy construction of debugger applications. It consists of:

- *JVMDI (Java Virtual Machine Debug Interface)*, an interface used by debugger back-ends running in the same process as a JVM to read information from the JVM.
- *JDWP (Java Debug Wire Protocol)*, a communication protocol used to transfer information between a debugger and the JVM that is running the debuggee.
- *JDI (Java Debug Interface)*, which is a high-level (from a debugger implementation standpoint) interface that allows a debugger to access a debuggee running in another JVM. Sun's implementation of JDI uses JDWP and JVMDI.

Essentially, every JVM that supports the JPDA is already instrumented to provide a lot of information for debugging purposes. Also, the compiler can produce extra debugging information that JPDA can make use of, such as mappings between bytecode and source code positions.

JDI provides support for examining the state of the running program fully (objects, classes and stack) and also modifying the state of the program on the Java program level; examining the actual memory content and low-level JVM state (such as the operand stack) is apparently not possible. It also supports debugger-controlled object instantiation and method invocation. One possible problem with the JDI is that it does not appear to support a convenient way of keeping track of all instances of a specified class. It also does not provide a way to execute parts of methods or watchpoints for local variable or array element accesses. It also may or may not support stepping one bytecode operation at a time; this behaviour is implementation-dependent, although Sun's JVM appears to step one bytecode at a time when asked to execute a minimum step. However, unlike any instrumentation-based methods, JPDA can also track accesses to Java objects performed by native methods. [71]

If operations can be reliably executed one at a time, this can be used to trace the executed operations, in which case explicit notifications for variable modifications, object creation and other interesting events are only needed for native methods. In order to work properly despite native methods and JVMs that may execute several operations in a single step, the system must be able to detect situations where operations have been executed unnoticed and check all the data structures in the program.

Using the JPDA is an easier and more portable solution than modifying the JVM. For this reason, most Java debuggers use the JPDA either directly (e.g. JDB) or indirectly (e.g. DDD and GVD use the JPDA through JDB [62, 63, 67]).

JPDA does not appear to provide any mechanism to list the classes available for loading. However, it does provide access to the class path used by the system class loader. These directories can then be scanned using standard directory listing techniques. Classes can then be loaded by invoking the system class loader on the debuggee using JDI method invocation (`java.lang.ClassLoader.getSystemClassLoader().loadClass(<class>)`).

Visualisations that require exact knowledge of the execution of the program, such as dynamic dependence graphs, cannot be implemented using JPDA unless source code or bytecode instrumentation is used in addition. This is described in more detail in Subsection 7.1.5.

7.1.5 Hybrid debuggee connection

The only instrumentation technique surveyed here that provides access to all the information needed for a visual testing tool is JVM instrumentation. However, this technique is unpractical due to maintenance and licensing issues. As the techniques surveyed here have different weaknesses, combining them may lead to a satisfactory solution.

The problem with concurrent reads and writes to variables disappears if the debuggee is suspended during the interval between a variable read or write and the corresponding message to the debugger. This can easily be implemented using JPDA in at least three ways:

- The read/write operation is actually performed by the debugger instead of the debuggee. In other words, the instrumentation process replaces variable reads and writes with read/write requests to the debugger.
- The debugger suspends all other threads while the read or write is being performed. This has the advantage that the original variable operations can be left in the code.
- The read/write operation is detected by JPDA and JPDA notifies the debugger before anything else happens in the JVM. Unfortunately, this solution only works for fields.

Tracking execution of uninstrumented code

Instrumenting the code during or after compilation provides a very accurate picture of what the instrumented code does. JPDA generates messages for all object field reads and writes and method calls, including those made in uninstrumented code. This is enough to narrow down the modifications made by a native or uninstrumented method somewhat.

Specifically, the set of data input to an uninstrumented method is a subset of the union of the following sets:

- The method's parameters (P).
- Values of accessed object and class fields (F).
- Any value reachable from an array reference belonging to $P \cup F$.

Similarly, the set of data output by an uninstrumented method is a subset of the union of:

- Any modified object and class fields (M).
- Any new arrays (N).
- The method's return value (R).
- Any value reachable by array element accesses from an array reference in $P \cup F \cup M \cup N \cup R$.

Thus, the debugger need not check everything accessible from the arguments of the native method and static fields for modifications after a native method is executed. The additional information from JPDA thus prevents native and noninstrumented methods from corrupting the debugger's copy of the debuggee's state.

This is not enough information for accurate data dependence analysis of native and uninstrumented methods, even if they are treated as single operations. However, approximating the in and out data sets of the methods as above ensures that all dependencies involving a native method are shown. Some spurious input and output dependencies may be shown for array elements.

One of the problems with the above is that modifications to objects belonging to uninstrumented classes must also be tracked. This will slow the entire debugging process down and cause a lot of irrelevant information to be logged. This can be avoided by ignoring any changes made to objects belonging to uninstrumented classes. This means that most of the internals of library classes will not be tracked in any way, which should mitigate the performance loss noticeably. Library objects can be tracked using accessors and modifiers, which limits the logged changes to those visible to the user.

7.1.6 Evaluation

Combining the virtual machine instrumentation and control facilities made available through JPDA with bytecode instrumentation provides access to the debuggee that is almost as good as an instrumented JVM and has none of the associated licensing and maintenance problems. Until such time as proper instrumentation is available (through an extension of JPDA or a new API) in a JVM suitable for production use, the bytecode instrumentation and JPDA hybrid solution seems to be preferable.

7.2 Manipulation of program history

Being able to step backwards through the execution of a program (similarly to the way in which algorithm animation usually allows stepping backwards) would allow the user to track down bugs by backing up from a state known to be wrong (e.g. an exception) until the bug itself is reached.

If the execution history of the program is displayed as a call tree, a list of events or a timeline, the user can easily move around in the execution history by selecting items in the tree, list or timeline. Alternatively, the user may be given controls that allow him to step backwards and forwards through the history.

7.2.1 Animation based on logging

The most straightforward way to provide animated display of the execution of a program is to store all the information needed to show previous frames of the animation, and then simply trace back and forward through this. To keep data storage requirements down, only the differences between successive displays should be stored. For example, Matrix stores all assignment operations in a special animator object [29].

In the case of a debugger it is not trivial to decide what information should be stored. Ideally, all changes to the state of the program would be stored which would allow the user to look at every value of every variable. However, this could require a lot of memory in complex cases. Therefore, the stored values should be limited to those that are actually interesting to the user. For example, in most cases, the values of the private fields of library classes are completely uninteresting.

This approach is easier to implement. However, it prevents the user from stepping backwards, changing something and executing the program from the earlier point with the changed value; in other words, the history is read-only. On the other hand, using logging (as in RetroVue and Matrix, for example) causes less problems with garbage collection, as it ensures that all interesting values are copied.

Even though this approach is limited compared to fully reversible execution, it removes a lot of the trial and error from the usual process of finding an error, in which the user typically executes the program past the first manifestation of the error when debugging and has to restart execution. With logging, the user can rewind the data view to the earlier stage.

Convertors as described in Subsection 5.1.6 cause problems when logging is used to store the execution history instead of reverse execution. When logging is used, the debuggee method calls needed by the convertor must be made while the debuggee executes; a convertor cannot rewind the program and execute a method on past data structures. Therefore, all the data needed by the abstractions must be extracted while the program is running; I call the routines that do this *extractors*. Extractors are similar to convertors, and have the same interface (except for the ability to extract further information) and applicability properties as convertors.

7.2.2 Reverse execution

With the ability to reverse the execution of a program, the user could after backing up to a bug manually correct the data structures and rerun the commands after the bug to check that it has been correctly identified [3]. However, reversing the execution of a program in the general case requires reversing everything the program has done. One way to do this is to reverse each instruction on the machine level by constructing a reverse instruction for each instruction where possible and saving any information that is lost when an instruction is executed so that it can be restored when the instruction is reversed [3]. As long as the only thing affected is the state of the JVM, this is not unduly difficult [15]. However, as soon as the program starts interacting with anything outside the JVM (in other words, performs any I/O, even local file access), the situation becomes a lot more complex.

Implementing reverse execution fully would require checking every available I/O operation for side-effects that need to be reversed. Also, the desired semantics of reversing some actions (e.g. keyboard input) are not clear. For example, putting a character back onto a buffer when the read operation is reversed makes sense in some contexts (reading lots of text from standard input), but is confusing in others (e.g. network applications). Implementing reverse execution for software with more I/O capabilities than access to a simple file system would require the ability to reverse every resource that a program can affect, e.g. other processes on other machines.

Reverse execution is also quite hard to implement without modifying the machine on which the code is running. While it is not unrealistic to modify a Java Virtual Machine written in software, this is nonetheless a nontrivial task. Also, in order for reverse execution to work properly, everything that happens in the machine must be logged. This leads to very large execution logs. In conclusion, this approach seems unnecessarily complex and provides little additional benefits at this stage. Leonardo supports reverse execution, but it requires a specialised virtual CPU and a set of reverse system calls to do so.

7.2.3 Evaluation

Reverse execution can theoretically provide a minor improvement in execution control, but it has too many unsolved problems for practical use. Therefore, logging would appear to be the best way to access the history of a program.

Chapter 8

Tool designs

Using the techniques and approaches described in the previous chapters, a visual testing tool can be designed that should be (assuming it can be implemented well) able to meet all of the criteria specified in section 2.1 well (a rating of III). However, implementing such a tool within the scope of this thesis is quite unrealistic. For this reason, a limited prototype was implemented instead.

The prototype visual testing tool is intended to be a proof of concept of the central new aspects of visual testing:

- The ability to interactively test a program without writing extra code to run test cases.
- A graphical data view at a high level of abstraction for both current and past data.

Therefore, tasks that can be satisfactorily performed using existing tools have been left out of consideration. For example, static analysis of source code can be done using Source-Navigator and Bandera. For the same reason, fine-tuning aspects such as the user interface and graphical presentation has been left out of consideration. Also, visualisations that appear to be clearly less suitable for visual testing than other visualisations have been left out. Finally, implementing the execution history views is considered a task for later work.

The designs for the full visual testing tool and the prototype, including some comments on the reasons for the less obvious design choices, are described in Table 8.1.

Feature	Full visual testing tool	Prototype	Section
Data view:			
View primitives	Standard textual representation (graphical primitive representations are less practical)	Standard textual representation (easy to implement, necessary)	4.1.1
View arrays	Tables, plots and images (plots and images are probably useful when debugging programs with large numeric arrays)	Tables (plots and images are only useful with large numeric arrays and are widely available)	4.2
View classes and objects	Nested boxes, nested arrows	Nested boxes, nested arrows	4.3
View execution stack	Object-like view (easy to use)	Object-like view (easy to implement)	4.5.5
Code view:			
View package hierarchy	Package tree	Package tree	4.4.4
Source code view	Text with execution tracing	Text with execution tracing	4.4.1, 4.5.1
View the inheritance hierarchy	Class diagram	Not implemented (in e.g. BlueJ [27])	4.4.4
View reference graphs	Browsable reference graph	Not implemented (in e.g. Source-Navigator [77])	4.4.5
Execution history view:			
View call tree	Call tree with all operations	Not implemented (left for later work, in e.g. RetroVue [14])	4.5.6
View hybrid diagram	View branch of call tree as hybrid diagram	Not implemented (left for later work)	4.5.9
Dynamic dependence graphs	Browsable DDGs	Not implemented (left for later work, some unresolved practical issues)	4.5.11
Move back and forward in the logged execution history	Movement in steps or searching for specific events	Movement in steps or searching for the last modification to a variable (easy to implement and useful)	7.2
Elision and abstraction:			
Specify abstract data type and graphical representation for classes	User-defined abstraction based on data structures recognised by class/interface using accessors and modifiers	User-defined abstraction based on data structures recognised by class/interface using accessors and modifiers	5.1.4, 5.1.1, 5.1.2, 5.1.6
Automatic data elision	Hiding of data in large and library objects and arrays, string truncation	Hiding of library object data, string truncation (prototype not intended for use with large arrays)	5.2.1
Manual data elision	Show/hide individual objects and classes and all instances of a class or all classes in a package	Show/hide individual objects and classes and all instances of a class or all classes in a package (necessary for larger programs)	5.2.2
Code elision	Package tree (with methods), method list boxes, block collapsing	Package tree (other methods widely available)	5.3
Execution history elision	All techniques in Section 5.4	Not implemented (nothing to use it on)	5.4
Data editing:			
Edit primitive data values	Textual editing (easiest to use)	Textual editing (easy to implement)	6.1.1
Edit object references	Dragging and dropping of arrow ends or object boxes (easiest to use)	Dragging and dropping of arrow ends or object boxes (easy to implement)	6.1.2
Execution control:			
Execute methods (including constructors)	Pop-up method menu and invocation parameter object	Pop-up method menu and invocation parameter object	6.2.2
Start and stop execution of threads	Selected in data view	Selected in data view	6.3
Step through the execution of a thread	Step to next line, next instruction, into called methods, out of the current method and over method calls	Step to next line (most commonly used step, others can be added later)	6.3
Set and remove breakpoints	Selection of source code lines	Selection of source code lines	6.3
Set and remove watchpoints	Selection using data view	Not implemented (easy to add later, breakpoints can usually be used)	6.3

Table 8.1: Designs for a full visual testing tool and a prototype

Chapter 9

Prototype

In order to evaluate the most important new ideas in visual testing, I have produced a prototype, which I call *MVT* (*Matrix Visual Tester*).

MVT can be divided into the following parts, each of which corresponds to a package in the MVT package `fi.hut.cs.mvt`:

- Instrumentation of debuggee (`fi.hut.cs.mvt.connection.instrumentation`).
- Connection to debuggee (`fi.hut.cs.mvt.connection`).
- Data model (`fi.hut.cs.mvt.data`).
- View model (`fi.hut.cs.mvt.view`).
- User interface (`fi.hut.cs.mvt.ui`).

Each of these parts will be separately described in this chapter.

As its name suggests, MVT is implemented using Matrix. Matrix provides MVT with a framework for logging and visualisation of data structures. By using an existing visualiser, the time to produce the prototype visual testing tool was cut down noticeably. However, many of the execution history visualisations would have required extensive additions to Matrix.

The structure of MVT is shown in Figure 9.1.

9.1 Connection to debuggee

MVT uses the hybrid debuggee connection described in Section 7.1.5. Thus, it consists of instrumentation code and a runtime connection based on JPDA.

9.1.1 Instrumentation

In order to gather information effectively from the debuggee, the Java bytecode of the debuggee is instrumented with calls to several empty dummy methods (in the MVT debug call class `fi.hut.cs.mvt.connection.instrumentation.DebugCalls`) that take the information to be extracted as parameters. Calls to these methods can then be detected as method entry events using JPDA.

The instrumentation is implemented using the class file parsing and modification facilities of BCEL [17]. Before starting the debuggee, the user must tell the instrumenter which packages or classes to instrument. The instrumenter then adds the extra method calls to the debuggee class files.

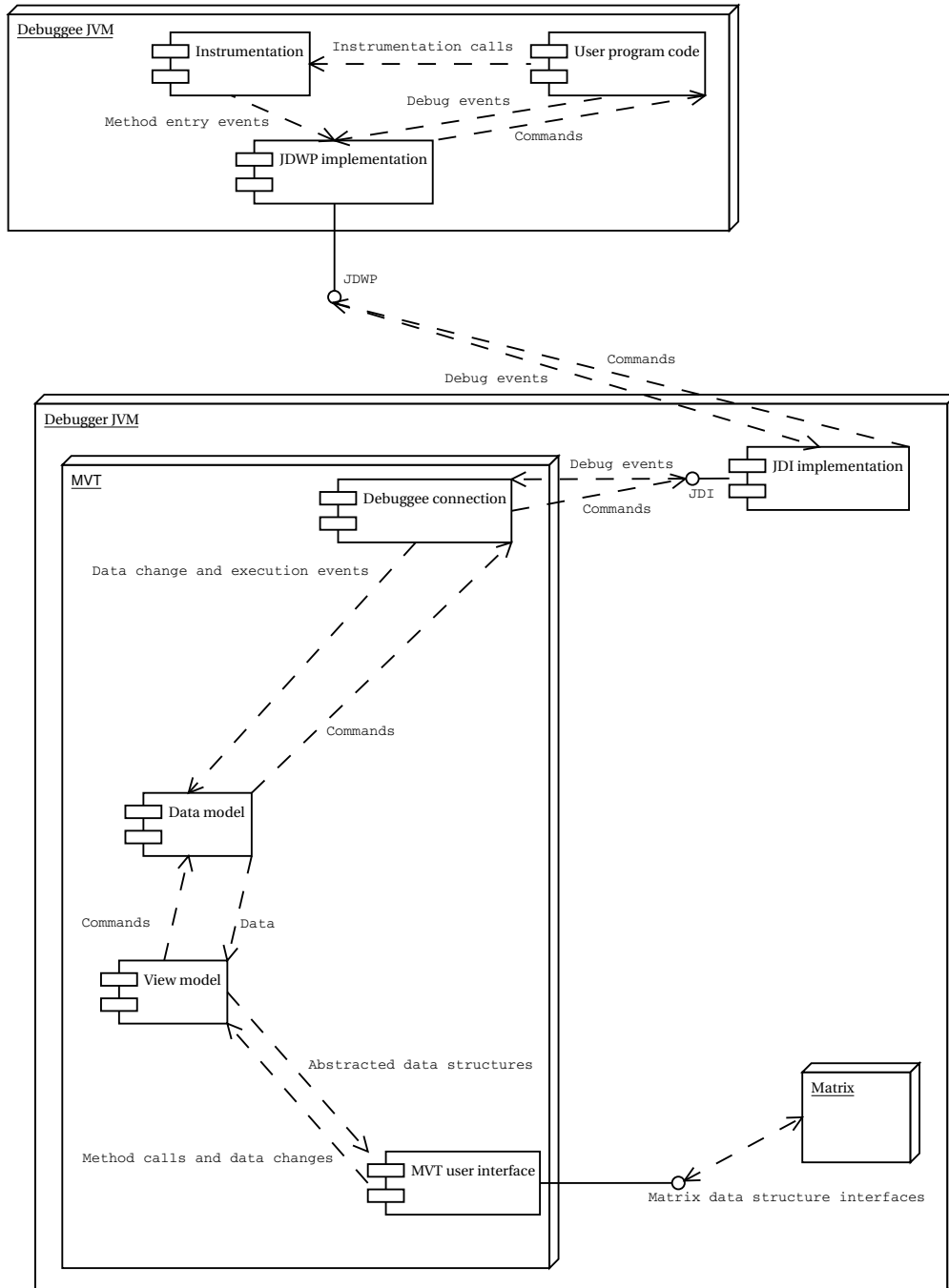


Figure 9.1: Structure of MVT

When the instrumenter instruments a class file, it adds instrumentation code before and after almost every bytecode instruction in the class file. Before each instruction, code is added to copy the values read from the operand stack by the instruction and call a suitable method in `DebugCalls` to send the values to the debugger. Similarly, after the instruction, if the instruction put any new values on the stack, these values are copied and passed to a method in `DebugCalls`.

The instrumenter also adds code to the start of every method to generate a unique identifier for each method invocation. This is used by the data model to identify stack frames uniquely.

The instrumentation has a few minor problems that are related to the requirements placed by the JVM on bytecode. References to newly created objects that have not been initialised can only be manipulated in a limited number of ways. In particular, passing them as parameters to any method other than a constructor for the right object type is considered an error. Due to this, newly created objects will not be visible until the first field value is written to them by instrumentation code or a reference to the object is stored in an object. In practice, the only effects of this limitation are that new objects are not visible until the instrumented code actually does something with them. This may be mildly confusing, but no relevant information is lost.

Adding the instrumentation calls may cause a method to expand beyond the 64 kB limit enforced by the JVM. Methods written by human programmers are seldom large enough to cause problems. However, some problems may be experienced with machine-generated code. No problems of this type were observed during testing.

9.1.2 Runtime debuggee connection

All communication with the debuggee, including starting the JVM in which the debuggee is run, is done using JPDA [71]. MVT uses JDI to access JPDA features.

Initially, the debuggee is started using the normal JDI procedure. In order to remove the need to start a `main` method immediately, the debuggee VM is started using a dummy class with a `main` method that does nothing except run into a breakpoint.

JPDA only allows one debugger-initiated method invocation to be active at a time. If the debugger invokes a method through JPDA while a method invocation is active, JDI cryptically reports an unknown JDWP error of type 502 (labelled `ALREADY_INVOKING` in the JDWP specification). To circumvent this problem, MVT does not invoke the methods the user specifies directly. Instead, it uses the thread created for the dummy `main` method to start a new thread in which the user-specified method is invoked using the Java reflection API (`java.lang.reflect`). This allows the user to invoke methods at any point in the debuggee's execution. Accessors are run in separate threads in a similar fashion, which enables MVT to kill an accessor that has not finished within a suitable time limit.

MVT allows the user to step through running code a line at a time or run it to the end of a user method invocation or a breakpoint.

The debuggee connection is based on an event queue mechanism similar to that of JPDA. The debug connection waits for events from JPDA and then processes them. In processing a JPDA event, the debuggee connection may generate one or more events of its own, which it put on its own output queue. These events are similar to the events produced by JPDA, but they are at a slightly higher level and the events produced by instrumentation calls have been converted into a more sensible form. The data model part of MVT waits for events from the debuggee connection part, and updates its own data model correspondingly. Whenever the user wants the debuggee to continue or step, the data model waits for and processes events until the criterion for interrupting execution is satisfied.

Due to namespace conflicts caused by the classes used by the instrumentation, MVT cannot be used to debug programs that use the package `fi.hut.cs.mvt` or any package inside it. In practice, the only problem this causes is that MVT cannot examine itself unless the package names are changed in the copy that is examined.

9.2 Data model

The data model is essentially a log of all changes made to the data structures in the program. Other aspects of the execution history, such as invoked methods, can also be considered a part of the data model. Currently, only operations that affect the data in the debuggee are logged.

Whenever an event is received from the debuggee connection, the corresponding changes are made to the data model. MVT stores the data model in memory structures provided by Matrix (`matrix.structures.memory`). These structures automatically log changes and allow the user to step back and forth through the history of the data structures using the animation controls provided by Matrix.

The data model consists of data elements (`DataElement`), which can be either data containers (`DataContainer`) or model variables (`ModelVariable`). Model variables correspond to individual variables in the debuggee: fields, local variables and array elements. Data containers correspond to threads (or their stacks), stack frames, objects, classes and arrays. A model variable always contains a JDI reference to a value in the debuggee. Similarly, each data container contains a reference to the corresponding thread, stack frame, object, class or array in the debuggee. Data containers also contain a number of named elements. Each element is a model variable. JDI provides references to objects, classes, threads and arrays. Stack frames are identified using a identifier generated by the instrumentation code.

The data model is also responsible for detecting possible changes to data values extracted using accessors and running the extractors to update the values. The data model keeps track of the extracted values. If an object is known to be mutable and updating extracted values is desired for the object's class, extracted values are updated whenever a method call on the object exits. For most library classes (e.g. `Collections`), this is sufficient.

9.3 View model

The view model consists of data structures that can be visualised by Matrix (they implement various FDT interfaces in `matrix.structures.FDT`) that can update themselves to reflect the information in the data model. The top level view model object is called the nested object graph, which implements an extended version of the Matrix graph interface. The nested object graph contains container arrays that correspond to the data containers in the data model. The container arrays contain named elements that are data structures that can be visualised by Matrix. The elements can be e.g. primitive variables, other container arrays and arrows to other container arrays.

Abstraction and elision are handled by the view model. When the view model is updated based on the data model, convertors are applied to transform the data into the desired form.

9.4 User interface

The user interface allows the user to access all the features of MVT. Most of the MVT user interface is provided by Matrix. The MVT main window is an extended version of the Matrix prototype user interface. It contains a visualisation of the nested object graph based on the data model, a package tree and a view of the source code of the currently running code.

Operations that manipulate the code of the debuggee and visualisation settings of classes and packages are performed using the package tree. To preserve screen space, the branches of the tree corresponding to packages are collapsed by default. The class visualisation settings are the default settings for the instances of the class; object-specific settings override

these. If a class does not have a value for a setting, it inherits one as described in Subsection 5.1.6.

The source code view shows the line at which the debuggee has stopped highlighted in the source code.

The nested object graph shows the data in the debuggee and allows the user to manipulate the debuggee by modifying data through drag and drop or pop-up menus.

The visualisation settings for objects include:

- Transformations for the container array view such as flipping or rotating the array as well as turning indices and title bars on or off.
- The name of the (possibly virtual) field to show the object as.
- Whether the object is shown even though no reference to it is visible.

The class-specific default object visualisation settings include:

- Defaults for all of the object-specific settings.
- The extractors and convertors for instances of the class. The extractors and convertors are Java classes that implement the `fi.hut.cs.mvt.data.Extractor` or `fi.hut.cs.mvt.data.Convertor` interface. When defining the extractor or convertor to use for a class, `String` and `DataContainer` parameters may be specified to configure the extractor or convertor for use with the class.

The class-specific defaults can be saved between sessions. Object-specific settings cannot be saved.

Figure 9.2 contains a screen shot of MVT taken during the execution of the hash table test case (see Section 10.2). From the top down, the MVT/Matrix window in the screen shot contains:

- Menus that contain global settings and commands.
- The animator controls used to step through the logged execution history.
- The debuggee state view, containing (from left to right):
 - The active thread, containing a stack frame, which contains the local variables for the method invocation. If several method invocations are active in a thread, they are arranged vertically in a table inside the thread box. If several threads are active, they are all shown (unless the user changes this using elision control).
 - A `LinearHashTable` object. Both its fields (the actual table, and its size) are shown, as well as the result of passing the table to the `valueOf` method in the Java `String` class. As only the keys in the table are interesting in this context, only the key (an `Integer` object) is shown for each hash table entry.
 - The `LinearHashTable` class, which contains two static fields.
- The tree of packages and classes in the class path. All packages are collapsed to save space.
- The search method invocation described in the use case description, before it is executed using its pop-up menu.

Figure 9.3 contains a screen shot of MVT taken during the execution of the XML tree test case (see Section 10.3). The attributes and children of each XML node are shown.

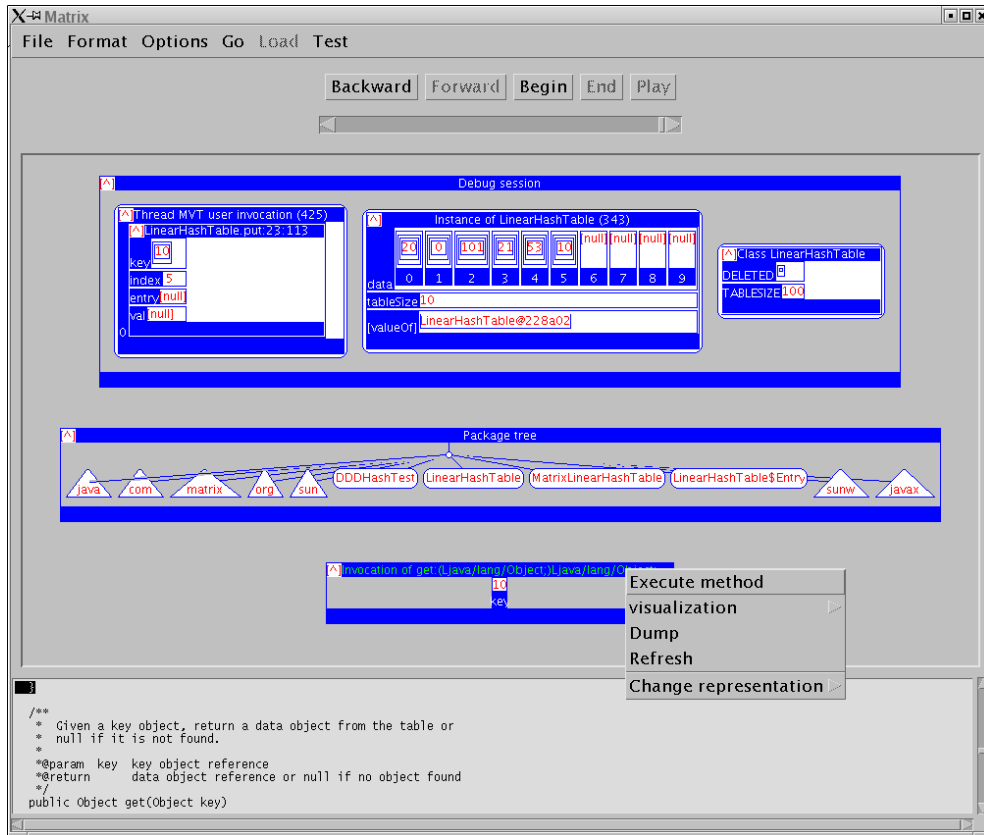


Figure 9.2: MVT running the hash table test case

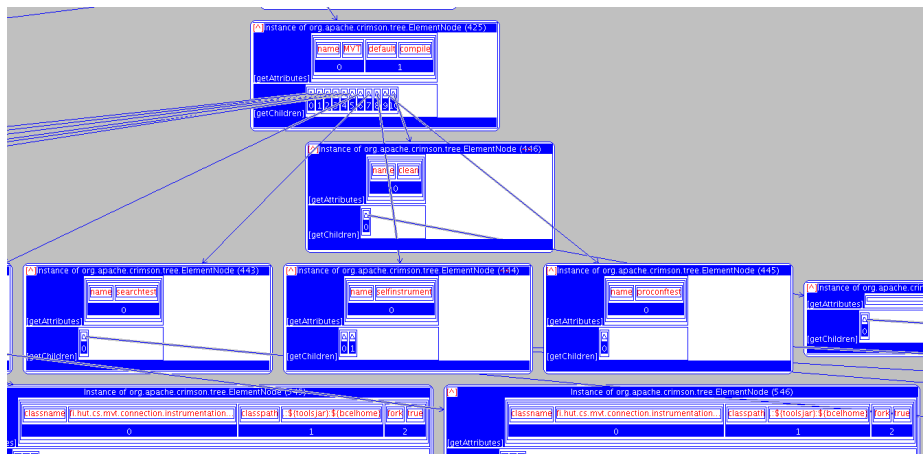


Figure 9.3: Part of an XML tree shown in MVT

Chapter 10

Use cases

In order to provide examples of the use of a visual testing tool and some concrete cases to evaluate the prototype against, I will present a few use cases for a visual testing tool. These test cases are intended to demonstrate that a visual testing tool can be used as a debugger to examine a wide range of programs and in addition provide a clearer data view, decrease the need for test code and provide a better way to examine the execution history of the program.

The first three test cases (examining a sort routine, a hash table and a library data structure) are intended to simulate different unit testing scenarios, as this type of use is particularly conducive to interacting testing. Also, keeping the tests small makes the tests easier to perform. The small tests concentrate on testing the presentation of the program state, the prototype's support for causal understanding and the generality of the prototype when applied to different data structures. The hash table test is also intended to test the data modification and execution control aspects of the the prototype. The library data structure test should demonstrate the prototype's suitability for one of the most useful applications of a more abstract data view. The last test case tests the suitability of the prototype for the testing of larger programs. These four tests should provide an indication of the suitability of the prototype for software testing without requiring extensive testing that can not be performed within the scope of this thesis.

10.1 Debugging a sort routine

Debugging a bubble sort routine is a simple test that is within the capabilities of all of the surveyed software visualisation systems. A bubble sort routine with an off-by-one error in a loop condition is run on a small unsorted integer array and examined with a software visualisation tool in order to find the bug. The sort routine is as follows (spotting the error is left as an exercise for the reader):

```
public class BubbleSort {
    public static void sort(int[] table) {
        for(int i=0;i<table.length;i++)
            for(int j=i+1;j<table.length-1;j++)
                if (table[i]>table[j]) {
                    int temp=table[i];
                    table[i]=table[j];
                    table[j]=temp;
                }
    }
}
```

This use case is usually easy to handle with existing graphical and visual debuggers, as they can all display integer arrays in a meaningful fashion. In most debuggers and program visualisers, all that has to be done is to load the program and step through it displaying the state of the array. In order to do this with algorithm animation and simulation tools, the code must be modified or extended to conform to the interfaces used by the tools. In Lens and Leonardo, a visualisation must be constructed.

For simple programs such as this, single stepping is quite adequate. However, being able to step backwards through states of the program and list the executed operations may also be useful, as it eliminates the need for stepping through the program repeatedly.

In this case, a visual testing tool can be used much like a logging debugger such as RetroVue.

Sequence

- Run the broken sort routine on the array $\{4, 7, 3, 2, 5, 1\}$, stepping through execution and viewing the array and loop indices.
- Step backwards through the execution of the program, if possible. The array and indices should be visible.

10.2 Testing a hash table

This use case demonstrates the ability of a software visualisation tool to display structures consisting of objects linked with references at a suitable level of abstraction. It also provides an example of a problem that is easily detected using visual testing that is hard to find using normal unit testing methods.

A hash function unsuited to the data (the key modulo 10) causes hash tables that use linear probing to become unevenly loaded and therefore inefficient. A small table is constructed and visualised in order to find this inefficiency. The linear probing hash table implementation at [74] is used for this test.

When using a software visualiser with no execution control or simulation capabilities, test code must be written that creates a table and performs various operations on it. The test code must be carefully planned to cover all different cases or edited, recompiled and rerun repeatedly in order to provide a semblance of interactivity.

When using a debugger, the test code approach can be used. However, if the debugger supports method invocation, interactive testing can be performed by starting the program in the debugger and then invoking the hash table implementation's methods to construct a table and manipulate it. In all of the surveyed debuggers, this approach had some problems. BlueJ is the only debugger examined here that allows graphical method invocation; working with object references in the others is somewhat inconvenient. DDD and GVD can display the resulting structure as reasonably sensible tree of objects. With BlueJ and RetroVue, the user has to do some digging in the object viewer to extract the keys.

The program visualisers with programmable views and the algorithm simulation and animation tools again require additional work to produce a visualisation. However, this visualisation should be clearer than that produced by a debugger. An algorithm simulation tool also provides easy interactive testing.

In this case, a visual testing tool can provide the convenient interactivity and clear visualisation of an algorithm simulation tool without the extra adaptation work.

Sequence

- Insert the keys $\{20, 0, 101, 21, 53, 10\}$ into the hash table.
- Step through (line by line) searching for the key 10 in the hash table.

- Delete 10 and 0.
- Step through inserting the key 10 into the hash table.

When stepping through a search, the table entry that is being examined should be indicated.

10.3 Examining a data structure through a library API

This use case is intended to demonstrate how complex library data structures written to conform to a published API can be visualised effectively.

As an example, an XML document will be loaded, parsed and accessed using the DOM API (as described in [57]) and visualised using the visual testing tool. When using a debugger, the nodes in the DOM tree are shown in a very implementation-specific fashion that is very hard to work with. The children of a node can be requested using the `getChildNodes()` method, which returns a `NodeList` that can be read using its `item(int index)` method. The data can sometimes be read directly from objects referenced from the node object, but this is implementation-specific and the fields may be empty if the parser defers parsing until an attribute is requested (like e.g. Apache Xerces 2.3 appears to do).

Visual debuggers and visualisers that rely on visualising the data fields of an object are obviously at a disadvantage in this case, as they will show the implementation of the tree nodes instead of the tree data and structure.

The programmable program visualisers and algorithm animation and simulation tools again require some extra code to produce the visualisation, but they can produce a very clear view of the data.

With a visual testing tool, convertors can be combined to call the API methods and convert the values returned by them into a meaningful visualisation. If the API is very non-intuitive, some additional convertor code may be necessary. However, the amount of code needed should be less than the amount required to use an algorithm simulation tool.

Sequence

- Load the Matrix default configuration file (1896 lines of XML at the time of writing) using the DOM API.
- View the tree.

10.4 Studying the behaviour of a large program

This use case is intended to evaluate the suitability of the visual testing tool for work with large programs. As an example, I will use *jEdit* [69], a Java-based text editor freely available in source form. *jEdit* is a quite a large program (roughly 75000 lines of code). It is therefore reasonable to assume that any tool that can be successfully used to examine *jEdit* can also be used to examine other large programs.

Studying the static aspects of a program is seldom the most effective way to gain an understanding of how the program works. Running the program and trying out different features is often more educational even without a view into the internals of the program.

While manipulating a program using its normal user interface can give a good picture of the feature set of the program, it is harder to gain an understanding of how the program works this way. For this, some sort of view of the program execution and data is very useful. Being able to execute methods and modify data at will may help, but it is not necessary.

Sequence

- Start jEdit.
- By examining history and/or placing breakpoints, determine how jEdit determines which files are loaded on startup (where this setting is stored and how).

Chapter 11

Evaluation of prototype

The prototype will be evaluated in two ways. First, a feature set comparison will be used to compare it to the other surveyed tools. Second, the prototype will be tested against the use cases in Chapter 10.

11.1 Feature set comparison

When evaluating using the system defined in Table 3.1, MVT gets the results shown in Table 11.1. These results can be compared with the scores in Table 3.2. MVT was written with these criteria in mind, so it is hardly surprising that it scores at least \equiv in every category.

11.2 Evaluation using use cases

Evaluating MVT based on its feature set, as in Table 11.1, may not tell the whole story. In order to evaluate the performance of MVT, I have tested MVT in the use cases defined in Chapter 10. The performance of MVT in these use cases should give some indication of whether MVT is suitable for debugging.

To put MVT's results in perspective, the same tests will also be performed using DDD and Matrix. These two were chosen from the surveyed tools as they are freely available, can visualise Java programs, have high scores in Table 3.2 compared to similar tools and have at least limited support for everything required by visual testing.

Like most algorithm animation/simulation tools, Matrix cannot extract information about the execution flow of a program (e.g. stack frames and execution position) unless the program explicitly maintains data structures corresponding to this information and updates it or an extension to Matrix is written that connects to the program using instrumentation

System	Generality	Completeness	Data modification	Execution control	Representation	Abstraction	Automatic view control	Manual view control	Causal understanding
MVT	\equiv	\equiv	\equiv	\equiv	\equiv	\equiv	\equiv	\equiv	\equiv

Table 11.1: Feature set evaluation of MVT

Score	Meaning
-	The visualisation is completely useless or no visualisation was produced.
I	Some information is missing from the visualisation or it is hard to understand.
II	The visualisation is clear but has some problems.
III	The visualisation is clear and has no noticeable problems.

Table 11.2: Scoring system for evaluation of clarity of visualisations in test runs

and/or JPDA (like MVT does). For this reason, Matrix will be left out from tests where the amount of extra or modified code required to use Matrix would probably exceed 500 lines.

For each use case, the effort needed to prepare the desired visualisation will be measured using:

- The total time used by me when performing the test. This time includes the time needed to write any test code that is required. To avoid biasing the measurements against the first tools to be evaluated, no test code was reused between test cases, and the data structures to be visualised were studied before the first test started.
- The amount of lines of code (LOC) written or modified.

A large amount of lines of code should correspond to a low generality rating (the program must be modified or interface code written). If a lot of time is needed, the tool may be lacking in generality and/or automatic view control.

Similarly, for each test the clarity of the resulting visualisation will be evaluated according to the scale in Table 11.2. This depends on representation, abstraction and/or manual view control, and should directly affect how well the programmer can understand the visualisation. However, this evaluation may be somewhat subjective.

11.2.1 Debugging a sort routine

In addition to the general evaluation criteria, stepping backwards through the execution history is tested. This can be considered a test of the tool's support for causal understanding.

DDD

In order to perform this test with DDD, I had to write a simple `main` method that creates the table to sort and calls the sort routine. I also had to add all of the array elements individually to the visualisation due to technical problems with JDB (`oldjdb` in Sun Java 1.2.2 and 1.3.1 on i386 Linux). The resulting view was quite acceptable. I used DDD's "undo" feature to step back through the execution history.

Matrix

In order to perform this test with Matrix, I had to rewrite the bubble sort routine to implement the `Matrix Array` interface and store data in a `Matrix VirtualArray`. Most of the time was spent writing this code. Running and visualising the bubble sort was then a straightforward matter of putting the class files in the Matrix directory, opening them in Matrix, executing the sort routine and stepping through the resulting animation.

MVT

Running this test with MVT consisted of running the instrumenter on the bubble sort routine, loading MVT, creating the array, loading the bubble sort class, executing the bubble

System	Time	LOC	Clarity	Reverse stepping
DDD	10 min	6		Yes
Matrix	40 min	64		Yes
MVT	3 min	0		Yes

Table 11.3: Test results in bubble sort test

System	Time	LOC	Clarity
DDD	24 min	15	
Matrix	74 min	94	
MVT	16 min	0	

Table 11.4: Test results in hash table test

sort on the newly created array, and finally stepping through the result. No additional code was needed.

Summary

Table 11.3 contains the test results for the bubble sort use case. Using DDD and MVT was quick and satisfactory, while using Matrix required a disproportionate amount of extra code.

11.2.2 Testing a hash table

DDD

In order to perform this test with DDD, I had to write a simple `main` method that creates the table and calls the insert, delete and search routines. I also had to add the array elements individually to the visualisation due to technical problems with JDB and in order to get the desired view. The resulting view was acceptable.

Matrix

In order to perform this test with Matrix, I had to extend the hash table to implement the `Matrix Array` interface and modify it to store data in a `Matrix VirtualArray`. Most of the time was spent writing this code. Running and visualising the hash table was then a straightforward matter of putting the class files in the Matrix directory, opening them in Matrix, activating the test code to modify the table and stepping through the resulting animation.

MVT

Running this test with MVT was similar to the bubble sort test; instrumentation followed by loading the class into MVT and interactively testing in MVT. No additional code was needed.

Summary

Table 11.4 contains the test results for the hash table use case. Using MVT was reasonably painless, DDD required a bit more work but not too much, while Matrix required quite a lot of extra test code.

System	Time	LOC	Clarity
DDD	56 min	55	
Matrix	49 min	78	
MVT	84 min	32	-

Table 11.5: Test results in Matrix configuration file tree test

11.2.3 Examining a data structure through a library API

In this test, an XML tree accessed should be loaded using the DOM API and visualised as a tree. Each node in the DOM tree should be shown as a box containing the attribute type name, the attributes of the node (name and value of each attribute) and links to the child nodes. The order of the child nodes should be clearly visible.

As the purpose of this test is to examine how well MVT can be adapted to a new library API, the time used to write the extractors for XML data is included in the test.

DDD

In order to perform this test with DDD, I had to write a small program that loads the XML file, parses it using the DOM API and constructs a tree of simple objects containing the relevant data.

DDD failed to visualise even the converted tree properly. Apparently, DDD or `oldjdb` has problems with empty arrays and arrays with more than three elements. This caused most of the nodes of the tree to be scrambled and lack some information. This test strongly suggests that DDD is not suitable for production use.

Matrix

The XML test code for Matrix was similar to the code used with DDD, except that the parsed tree was made available to Matrix using the `Matrix Tree` interface.

The resulting tree representation should have been quite clear, but rendering problems (including a large black box that covered the entire tree and the right-hand side of the tree being cut off) made the tree hard to examine effectively.

MVT

Defining the DOM tree visualisation in MVT took roughly 40 minutes. This time mostly consisted of defining the abstractions for the DOM `NodeList` and `NamedNodeMap` classes (`Attr` objects could be directly visualised using extractors built into MVT). However, applying the DOM tree visualisation to the Matrix configuration file caused MVT to spend over half an hour extracting the tree before the JDWP implementation in the debuggee crashed. In other words, MVT produced no useful output. This indicates that MVT has severe performance and stability issues that make it unsuitable for use with large data structures.

Both the performance problems and the crash seem to be caused by Sun's JPDA implementation. The time between requesting a method invocation and the actual execution of the method invocation seems to increase with every invocation. JDWP reported an internal error before failing, which means that JDWP has a bug or insufficient error checking.

Summary

Table 11.5 contains the test results for the XML tree test. DDD and Matrix both required some test code and had problems producing a graph, even though a partial result was produced. MVT needed less test code, but failed to produce a graph due to implementation problems.

System	Time	Clarity
DDD	1 min	I
Matrix	3 min	II
MVT	5 min	II

Table 11.6: Test results in MVT build file tree test

As all of the debuggers experienced technical problems, I repeated the test with a smaller XML file (a pre-release version of the build configuration file for MVT, 94 lines) in order to determine whether the problems were related to the size of the file. Table 11.6 contains the results of this additional test. The time noted here is the time needed to apply the pre-defined visualisation to the new file. In this test, Matrix and MVT had no rendering problems and produced a usable result.

11.2.4 Studying the behaviour of a large program

This test is mostly about finding things in the execution history. If the tool used provides good support for causal understanding in practice, it is easy to find the right spot in the program. For these purposes, ease is considered equivalent to speed.

Using Matrix to examine the execution of a program of this type is quite unpractical, as it would require adding code to trace execution either involves instrumenting jEdit manually with code that keeps track of the execution position (which involves editing thousands of lines), writing an instrumenter that inserts the tracking code or adding a debugger connection to Matrix. The first option is extremely tedious, while the second and third duplicate parts of MVT.

DDD

Using DDD, I stepped through the code executed in the main jEdit class until I found the code that loaded the list of previously opened files and opened the listed files. This was quicker and easier than expected.

MVT

Even with only minimal extraction of data (`String` contents only) and instrumentation of only the main class (`org.gjt.sp.jedit.jEdit`), MVT used almost a GB of RAM even before jEdit had started completely. Apparently, the main class of jEdit performs enough operations on its own to create a huge log. Also, each attempt to run jEdit under MVT lasted for roughly ten minutes.

The problems in this test suggest that MVT collects too much data or stores it inefficiently and slows down the execution of a program badly. Thus, MVT is not suitable for testing large programs at once. When using MVT, the user must decide on a (small) part of a program and instrument and examine only that part.

Summary

MVT's extensive logging makes it unsuitable for use with large programs. DDD, on the other hand, actually seems to handle this type of use case well. The test results are in Table 11.7.

11.2.5 Evaluation

When examining a program with MVT, the user need only write extra code if he wants to define a new type of abstraction. In this respect, MVT provides a clear advantage over

System	Time
DDD	14 min
MVT	75 min (failed)

Table 11.7: Test results in jEdit examination test

Matrix, as the user does not need to explicitly define every type of object to be visualised. When using DDD, the abstraction must be added to the debuggee. When using Matrix, a class must be implemented that provides all of the visualisation.

11.3 Summary

While MVT has a feature set that is well suited for visual testing, it does not always work well in practice. In particular, MVT works very badly with programs that perform lots of operations or create large data structures. However, getting a good visualisation at a slightly higher abstraction level than the implementation level seems to be easier with MVT than with DDD or Matrix. This means that visual testing, even in this crude implementation, appears to simplify the testing of small chunks of code. In other words, MVT seems to be quite a decent tool for unit testing. MVT can also be used to generate visualisations of small programs quickly.

MVT has several performance-related problems that must be resolved in order to produce a good visual testing tool. In particular, more efficient ways to extract information from a debuggee and store the execution history are clearly needed.

Chapter 12

Conclusion

The goal of visual testing (outlined in Chapter 2) is to make it easier for programmers to test and debug their code. Visual testing does this by combining ideas from different types of software visualisation into a single tool that should make it easier for a programmer to try out different operations on program code and examine the results visually. I have evaluated several software visualisation tools in Chapter 3. All of these systems provide some features that could be useful in visual testing, but none of them is an effective visual testing tool.

Together, the surveyed software visualisation tools have almost everything that is needed for a visual testing tool. For this reason, I have examined and evaluated relevant visualisation (Chapter 4), elision and abstraction (Chapter 5) and debuggee control (Chapter 6) techniques. Most of these techniques can be found in the surveyed tools.

Using these evaluations, I have found ways to present the state and execution history of a program effectively. Most of the state of a program can be shown in a data view that primarily consists of objects (shown as boxes with data fields) that may contain or refer (through arrows originating in a data field) to each other. By augmenting the object display with boxes for classes and threads and adding a simple display of source code with highlighting of the current statement, the data view can display most of the state of the program. For clarity, data should be shown at a level of abstraction close to the level used by the programmer when writing his code. Making use of the abstraction inherent in object-oriented code seems to be an effective way to provide data abstraction in a debugger. Many ways to show the execution history of a program exist, but the best ways seem to be allowing the user to step through the states of his program and showing the executed operations as a tree or in the data view as a hybrid diagram.

In order to efficiently test software, the user should be allowed to specify, with a minimum of fuss, the operations he wants the software to perform. Allowing the user to graphically manipulate the data in his program and invoke methods at will provides additional flexibility and eliminates much of the need for test code.

Most of the aforementioned techniques have been used earlier, but they have not been combined. A visual testing tool consists almost entirely of software visualisation and debugging techniques that have been developed earlier and some additional ideas to make these different techniques work together.

After examining different approaches to the implementation of a visual testing tool in Chapter 7, I have presented (in Chapter 8) designs for a full visual testing tool and a prototype visual testing tool that demonstrates the feasibility of the visual testing concept and the new techniques applied in it.

Even though my tests (Chapter 11) show that the prototype visual testing tool has severe performance issues, my tests suggest that it is a useful tool for unit testing. As unit testing is one of the intended uses for a visual testing tool, this means that the prototype already goes some way towards the goals of visual testing.

MVT was put together in two man-months and contains only crude implementations of a few of the ideas of visual testing. Despite this, my tests suggest that it is an improvement over existing tools in tasks such as unit testing and debugging. This suggests that a well implemented complete visual testing tool can prove to be a powerful tool.

12.1 Future research

There are several matters that need to be researched in more depth before a proper visual testing tool can be produced:

- The execution history logs produced by MVT are far too large. Ways must be found to rid these logs of irrelevant and redundant information and store the remaining information efficiently.
- Some issues remain unsolved in adapting dependence graphs for use in visualisation.
- A more efficient way to extract information from a running program and control it is needed. In the long term, the best solution is probably to adapt a virtual machine to visual testing.
- The user interface of the visual testing tool needs to be planned more carefully than MVT was. In particular, common operations should be made easier to perform.
- The possibility of adding some sort of automatic checking of the validity of the program (such as assertions) combined with some sort of visible warnings in the visual testing tool should be investigated.
- The possibilities of adapting visual testing to languages other than Java should be investigated.

If the above matters are researched, it may be possible to produce a visual testing tool that is truly a useful aid in debugging and testing real-world software.

Bibliography

- [1] Hiralal Agrawal, Richard A. DeMillo and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Symposium on Testing, Analysis, and Verification*, pages 60–73, 1991.
- [2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, volume 25, number 6, pages 246–256, June 1990.
- [3] Tankut Akgul and Vincent J. Mooney, III. Instruction-level reverse execution for debugging. Technical Report GIT-CC-02-49, Georgia Institute of Technology, 2002.
- [4] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
- [5] Ricardo Baeza-Yates, Gastón Quezada and Gastón Valmadre. Visual debugging and automatic animation of C programs. In Peter D. Eades and Kang Zhang, editors, *Software Visualisation*, volume 7, pages 46–58. World Scientific, Singapore, 1996.
- [6] Ryan S. Baker, Michael Boilen, Michael T. Goodrich, Roberto Tamassia and B. Aaron Stibel. Testers and visualizers for teaching data structures. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 261–265. ACM Press, 1999.
- [7] Thomas A. Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, volume 29, number 4, pages 33–43, April 1996.
- [8] John L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, volume 4, number 1, pages 5–30, 1991.
- [9] Michael R. Birch, Christopher M. Boroni, Frances W. Goosey, Samuel D. Patton, David K. Poole, Craig M. Pratt and Rockford J. Ross. DYNALAB — A dynamic computer science laboratory infrastructure featuring program animation. In *Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, pages 29–33, March 1995.
- [10] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, volume 21, number 5, pages 14–36, May 1988.
- [11] Marc H. Brown. Zeus: a system for algorithm animation and multi-view editing. In *Proceedings of 1991 IEEE Workshop on Visual Languages*, pages 4–9, Kobe, Japan. IEEE, October 1991.
- [12] Marc H. Brown, Marc A. Najork and Roope Raisamo. A Java-based implementation of collaborative active textbooks. In *1997 IEEE Symposium on Visual Languages*, pages 372–379, September 1997.
- [13] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 177–186. ACM Press, 1984.

- [14] John Callaway. Visualization of threads in a running Java program. Master's thesis, University of California, June 2002.
- [15] Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, volume 45, number 6, pages 608–619, 2002.
- [16] Pierluigi Crescenzi, Camil Demetrescu, Irene Finocchi and Rossella Petreschi. Reversible execution and visualization of programs with LEONARDO. *Journal of Visual Languages and Computing*, volume 11, number 2, pages 125–150, 2000.
- [17] Markus Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, April 2001.
- [18] Matthew B. Dwyer, James C. Corbett, John Hatcliff, Stefan Sokolowski and Hongjun Zheng. Slicing multi-threaded Java programs: A case study. Technical Report SU CIS TR 99-7, Kansas State University, 1999.
- [19] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 9, number 3, pages 319–349, July 1987.
- [20] Martin Fowler and Kendall Scott. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley, Second edition, 2000.
- [21] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, June 2000.
- [22] Jyrki Haajanen, Mikael Pesonius, Erkki Sutinen, Jorma Tarhio, Tommi Teräsvirta and Pekka Vanninen. Animation of user algorithms on the Web. In *Proceedings of IEEE Symposium on Visual Languages*, pages 360–367. IEEE, 1997.
- [23] Robert R. Henry, Kenneth M. Whaley and Bruce Forstall. The University of Washington illustrating compiler. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 223–233. ACM Press, 1990.
- [24] Christopher D. Hundhausen. A meta-study of software visualization effectiveness. *Journal of Visual Languages and Computing*, volume 13, number 3, pages 259–290, 2002.
- [25] Bertrand Ibrahim. World-wide algorithm animation. In *Conference Proceedings of the First World-Wide-Web conference*, pages 305–316, May 1994.
- [26] Xiaoping Jia. *Object-oriented Software Development Using Java*. Addison-Wesley, June 2000.
- [27] Michael Kölling, Bruce Quig, Andrew Patterson and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology*, volume 13, number 4, December 2003.
- [28] Ari Korhonen. World Wide Web (WWW) tietorakenteiden ja algoritmien tietokoneavusteisessa opetuksessa. Master's thesis, Helsinki University of Technology, May 1997.
- [29] Ari Korhonen. *Algorithm Animation and Simulation*. Licentiate's thesis, Helsinki University of Technology, May 2000.
- [30] Ari Korhonen. *Visual Algorithm Simulation*. Doctoral thesis, Helsinki University of Technology, 2003.

- [31] Ari Korhonen and Lauri Malmi. Matrix — Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces - AVI 2002*, pages 109–114. ACM, 2002.
- [32] James F. Korsh, Paul S. LaFollette, Jr. and Raghvinder Sangwan. Animating students and programs in the laboratory. In *Proceedings of 1998 Frontiers in Education Conference*, Tempe, Arizona. ASEE and IEEE, November 1998.
- [33] Paul S. LaFollette, Jr., James F. Korsh and Raghvinder S. Sangwan. A visual interface for effortless animation of C/C++ programs. *Journal of Visual Languages & Computing*, volume 11, number 1, pages 27–48, February 2000.
- [34] Craig Larman. *Applying UML to Patterns*. Prentice-Hall, Second edition, 2002.
- [35] Bil Lewis. Debugging backwards in time. In Michiel Ronsse, editor, *Proceedings of the Fifth International Workshop on Automated Debugging*, Ghent, Belgium. September 2003.
- [36] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Second edition, April 1999.
- [37] Sougata Mukherjea and John T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 456–465. IEEE Computer Society Press, 1993.
- [38] Sougata Mukherjea and John T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction (TOCHI)*, volume 1, number 3, pages 215–244, 1994.
- [39] Brad A. Myers, Ravinder Chandhok and Atul Sareen. Automatic data visualization for novice Pascal programmers. In *Proceedings of IEEE Workshop on Visual Languages*, pages 192–198, Pittsburgh, Pennsylvania, USA. October 1988.
- [40] James Noble, Lindsay Groves and Robert Biddle. Object oriented program visualisation in Tarraingim. Technical Report CS-TR-95/15, Victoria University of Wellington, September 1995.
- [41] *Unified Modeling Language (UML), version 1.5*. Object Management Group, 2003.
- [42] John O’Connor and Edmund Robertson. The Arabic numeral system. In *The MacTutor History of Mathematics archive*. University of St Andrews, January 2001.
- [43] Rainer Oechsle and Thomas Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In S. Diehl, editor, *Software Visualization*, pages 176–190, Dagstuhl Castle, Germany. Springer-Verlag, 2002.
- [44] Willard C. Pierson and Susan H. Rodger. Web-based animation of data structures using JAWAA. In *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, pages 267–271, 1998.
- [45] Alex Potanin and James Noble. Checking ownership and confinement properties. Technical Report CS-TR-02/6, Victoria University of Wellington, June 2002.
- [46] Blaine A. Price, Ronald M. Baecker and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, volume 4, number 3, pages 211–266, 1993.

- [47] Richard Rasala. Automatic array algorithm animation in C++. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 257–260. ACM Press, 1999.
- [48] Gruia-Catalin Roman, Kenneth C. Cox, C. Donald Wilcox and Jerome Y. Plun. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, volume 3, number 2, pages 161–193, June 1992.
- [49] RTI. The economic impacts of inadequate infrastructure for software testing. Planning report 02-3, NIST, May 2002.
- [50] Raghvinder S. Sangwan, James F. Korsh and Paul S. LaFollette, Jr. A system for program visualization in the classroom. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 272–276, Atlanta, Georgia, USA. ACM, ACM Press, New York, New York, USA, 1998.
- [51] Saurabh Sinha and Mary Jean Harrold. Control-flow analysis of programs with exception-handling constructs. Technical Report OSU-CISRC-7/98-TR25, Ohio State University, July 1998.
- [52] John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, volume 23, number 9, pages 27–39, September 1990.
- [53] John T. Stasko. Using student-built algorithm animations as learning aids. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, pages 25–29, San Jose, California, USA. February 1997.
- [54] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, volume 18, number 2, pages 258–264, June 1993.
- [55] Thomas G. Stockham and Jack B. Dennis. FLIT — Flexowriter Interrogation Tape: A symbolic utility program for the TX-0. Memo 5001-23, MIT, July 1960.
- [56] *Inner Classes Specification*. Sun Microsystems, February 1997.
- [57] *Java 2 Platform, Standard Edition, v 1.4.1 API Specification*. Sun Microsystems, 2002.
- [58] Erkki Sutinen, Jorma Tarhio, Simo-Pekka Lahtinen, Antti-Pekka Tuovinen, Erkki Rautama and Veijo Meisalo. Eliot — An algorithm animation environment. Technical Report A-1997-4, Department of Computer Science, University of Helsinki, Finland, November 1997.
- [59] Fumiaki Umemori, Kenji Konda, Reishi Yokomori and Katsuro Inoue. Design and implementation of bytecode-based Java slicing system. Technical Report 407, Osaka University, December 2002.
- [60] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, volume 25, number 7, pages 446–452, July 1982.
- [61] Steven J. Zeil. *AlgAE (Algorithm Animation Engine) Programmers' Manual*. Old Dominion University, March 2001.
- [62] Andreas Zeller. Datenstrukturen visualisieren und animieren mit DDD. In *Proc. GI-Workshop: Softwarevisualisierung (SV 2000)*, Schloß Dagstuhl. May 2000.
- [63] Andreas Zeller. *Debugging with DDD*. Universität Passau, First (for DDD version 3.2) edition, January 2000.

- [64] Jianjun Zhao. Dynamic slicing of object-oriented programs. In *Technical Report SE-98-119*, pages 17–23. Information Processing Society of Japan, May 1998.
- [65] Peter Brummund and Ngozi V. Uti. Complete Collection of Algorithm Animations. WWW site at <http://www.cs.hope.edu/~algsanim/ccaa/>, June 2001. Checked on 2nd October 2003.
- [66] Nathan Fiedler. JSwat — Graphical Java debugger. WWW site at <http://www.bluemarsh.com/java/jswat/>, 2003. Checked on 2nd October 2003.
- [67] The GNU Visual Debugger. WWW site at <http://libre.act-europe.fr/gvd/>, July 2003. Checked on 2nd October 2003.
- [68] Javix visual debugger. WWW site at <http://www.javix.com/>, 2000. Checked on 2nd October 2003.
- [69] jEdit. WWW site at <http://www.jedit.org/>, September 2003. Checked on 2nd October 2003.
- [70] JIVE. WWW site at <http://jive.dia.unisa.it/>, 2003. Checked on 2nd October 2003.
- [71] Java Platform Debugger Architecture. WWW site at <http://java.sun.com/j2se/1.4.1/docs/guide/jpda/>, 2002. Checked on 2nd October 2003.
- [72] Chuck McManis. A first look at Borland’s JBuilder IDE. WWW page at <http://www.javaworld.com/javaworld/jw-11-1997/jw-11-indepth.html>, 2003. Checked on 2nd October 2003.
- [73] Prosasim. WWW site at <http://www.prosa.fi/eng/pr2Sim.htm>. Checked on 2nd October 2003.
- [74] Graham Roberts. Simple linear probing hash table class. WWW page at <http://www.cs.ucl.ac.uk/staff/G.Roberts/lb11/classlinearhashtable.html>, March 2002. Checked on 2nd October 2003.
- [75] Johannes Sixt. KDbg — A graphical debugger interface. WWW site at <http://members.nextra.at/johsixt/kdbg.html>, September 2003. Checked on 2nd October 2003.
- [76] Skillmarket — A daily look at in-demand tech skills. WWW site at <http://mshiltonj.com/sm/>, October 2003. Checked on 2nd October 2003.
- [77] Source-Navigator. WWW page at <http://sourcnav.sourceforge.net/>, June 2003. Checked on 2nd October 2003.
- [78] Tango/04 visual debugger. WWW site at <http://www.tango04.com/products/vdw/>, 2003. Checked on 2nd October 2003.
- [79] TIOBE programming community index. WWW site at <http://www.tiobe.com/tpci.htm>, October 2003. Checked on 2nd October 2003.
- [80] Introduction to the VC++.NET debugger. WWW page at http://csciwww.etsu.edu/blair/Using_Debugger.htm, August 2002. Checked on 2nd October 2003.
- [81] Visicomp. WWW site at <http://www.visicomp.com/>, 2002. Checked on 2nd October 2003.