DISSERTATION

Model-Based Debugging of `Java` *Programs Using Dependencies*

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Prof. Dr. Markus Stumptner

Institut für Informationssysteme, Abteilung für Datenbanken und Artificial Intelligence (184/2)

eingereicht an der Technischen Universität Wien

Fakultät für Technische Naturwissenschaften und Informatik

von

Dominik Wieland

9225211

A-1180 Wien, Pötzleinsdorferstraße 80

Wien, am 6.11.2001 _____

Kurzfassung

Die Technik der modellbasierten Diagnose wurde bereits für eine breite Palette an Diagnoseproblemen, die von der Diagnose rein physikalischer Systeme bis hin zur Fehlersuche in Computersoftware reichen, erfolgreich eingesetzt. In dieser Arbeit wird modellbasierte Diagnose dazu verwendet, Fehler im Quellcode von `Java` Programmen zu lokalisieren. Auf Basis des Diagnoseprozesses wird ein weiterreichender Debugging-Ansatz entwickelt, der den Benutzer während des Debuggings möglichst automatisiert unterstützt und so die Anzahl der Benutzerinteraktionen, die benötigt werden um einen Fehler eindeutig zu lokalisieren, minimiert.

Da ein Modell des analysierten Systems die Voraussetzung für die Verwendung modellbasierter Ansätze ist, behandeln Teile dieser Arbeit die Modellierung von `Java` Programmen. Im Einzelnen werden drei verschiedene Modellklassen definiert, die alle auf den funktionalen Abhängigkeiten des zugrundeliegenden `Java` Programms basieren (funktionale Abhängigkeitsmodelle). Die unterschiedlichen Modellklassen unterscheiden sich unter anderem in den Daten, die während der Modellierung verwendet werden, und ihrem jeweiligen Abstraktionsgrad. Es wird gezeigt, wie `Java` Systeme automatisch in funktionale Abhängigkeitsmodelle transformiert werden können und welche Eigenschaften und Unterschiede die einzelnen Modellklassen aufweisen.

In Folge wird im Detail beschrieben, wie funktionale Abhängigkeitsmodelle zusammen mit modellbasierten Diagnosealgorithmen dazu verwendet werden können, einerseits mögliche Fehlerstellen in `Java` Programmen zu berechnen (Diagnose) und andererseits einen bestimmten Quellcodefehler eindeutig zu lokalisieren (Debugging). Es wird die `JADE` Debugging-Umgebung vorgestellt. Dabei handelt es sich um eine Prototypentwicklung, die die in dieser Arbeit untersuchten Modellierungs- und Debugging-Ansätze realisiert. Dies beinhaltet eine Beschreibung davon, welche Benutzerinteraktionen notwendig sind und wie Quellcodefehler mit diesem Softwareentwicklungswerkzeug lokalisiert werden können.

Der `JADE` Debugger wurde anhand mehrerer `Java` Methoden getestet und auf seine Diagnose- und Debugging- Fähigkeit im Zusammenhang mit mehreren Quellcodestrukturen, unterschiedlichen Fehlerklassen und verwendeten Modellen untersucht. Alle erzielten Resultate sind ebenso Bestandteil dieser Arbeit, wie eine Diskussion über die Vor- und Nachteile der vorgestellten Ansätze. Weiters werden Möglichkeiten präsentiert, wie die `JADE` Debugging-Umgebung in ihrer Diagnose- und Debugging-Leistung in zukünftigen Versionen verbessert werden kann. Den Abschluß dieser Arbeit bilden eine Analyse der zukünftigen Rolle modellbasierter Debugging-Werkzeuge und eine Diskussion darüber, wie diese

in bestehende Softwareentwicklungssysteme integriert werden können. Erst die Verbindung zu einem Gesamtkonzept garantiert eine optimale Unterstützung des Benutzers in allen Phasen des Softwareentwicklungsprozesses.

Abstract

Model-based diagnosis has successfully been applied to a wide variety of diagnosis problems, ranging from purely physical systems to the software domain. In this work we use model-based diagnosis techniques to compute possible fault locations of buggy `Java` programs. Based on this diagnosis process, we present an iterative debugging approach, which is designed to guide a user through a debugging session in a maximum automatic way which minimizes the amount of user interaction needed to non-ambiguously localize a given source code bug.

Since the model-based approach is based on the existence of a model of the analyzed target system, part of this work deals with the creation of models of `Java` programs, which are suitable for debugging. In particular, we define three different model types, which make use of the functional dependencies of the underlying `Java` program (functional dependency models). The models differ in the amount of information used during their creation and their respective level of abstraction. We show how `Java` systems can automatically be transformed into a functional dependency model and discuss the various properties and differences of the resulting models.

We describe in detail how functional dependency models, together with standard model-based diagnosis techniques, can be used to compute bug candidates (diagnosis) and non-ambiguously identify individual bug locations in `Java` programs (debugging). We present the `JADE` debugging environment, a prototype debugger, which implements the modeling and debugging principles described in this work. We describe the various types of user interaction performed by the `JADE` system and show how it can be used to efficiently locate bugs in `Java` programs.

We test the `JADE` debugger on various `Java` methods in order to evaluate the diagnosis and debugging performance of the tool in the context of different source code structures, fault classes, and underlying model types. The results of all performed tests are stated and various advantages and drawbacks of our approaches are discussed. We present ideas how the `JADE` debugging environment can be improved in future versions in both, its diagnosis and debugging performance. Finally, we analyze the future role of model-based debugging tools and show how they could be incorporated into existing software development tools in order to provide an optimal support for the user during the whole software engineering process.

Acknowledgements

- My first thank you goes to the supervisors of this thesis, *Markus Stumptner* and *Franz Wotawa*. I am grateful to Franz Wotawa for introducing me to the scientific community and the principles of model-based diagnosis. Without his continuous support and innumerable discussions on the topics of this work, it would not have been possible to complete this thesis. I also very much appreciate the help and support of Markus Stumptner during the `JADE` project and his valuable comments on earlier versions of this thesis.

- I would also like to thank the head of our department, *Georg Gottlob*, for financing parts of the `JADE` project through the Wittgenstein Fund.

- Furthermore, I am very grateful to *Wolfgang Mayer* for proofreading this thesis and his very valuable comments on parts of this work. I would also like to mention the support of *Cristinel Mateis*, whose work in the course of the `JADE` project underlies parts of this thesis.

- Last, but not least, I would like to take this opportunity to thank my parents, *Gertraud* and *Elmar*, for all their love and support throughout the years. I sincerely appreciate what they have done for me in all aspects of my personal development.

Contents

I Theoretical Foundations **1**

1 Introduction **3**

2 Software Debugging **9**
2.1 Terminology of software anomalies 9
2.2 Classifying software faults 11
2.3 Fault taxonomies & statistics 13
2.4 Software debugging 16
2.5 Debugging techniques 18

3 Model-Based Diagnosis & Debugging **23**
3.1 Model-based diagnosis 23
3.2 Model-based debugging 27

4 The JADE Project **31**
4.1 The Java programming language 31
4.2 Goals of the JADE project 32
4.3 Current state of the JADE project 33

II Modeling **35**

5 Modeling Java Systems **37**
5.1 Software modeling 37
5.2 Java systems 40
5.3 Compile-time description of Java systems 41
5.4 Run-time descriptions of Java systems 43
5.4.1 The dynamic viewpoint 44
5.4.2 The static viewpoint 45

6 Functional Dependency Models **47**
6.1 Variable occurrences 47
6.2 Functional dependencies 49
6.3 Types of dependencies 50
6.4 Locations 51
6.5 Functional dependency models 53
6.5.1 FD expression models 53
6.5.2 FD statement models 53

6.5.3 FDMs of blocks and methods . . . 54
6.5.4 FDMs of host environments . . . 55
6.6 Model views . . . 55
6.6.1 Internal models . . . 55
6.6.2 External models . . . 57
6.7 Combining FDMs . . . 59
6.8 Properties of FDMs . . . 60

7 The *ETFDM* 61
7.1 FDs in *ETFDMs* . . . 61
7.2 Collecting the FDs of a single evaluation trace . . . 63
7.3 Combining multiple ETFDMs . . . 65
7.4 A complete FDM . . . 67
7.5 Handling aliasing with the *ETFDM* . . . 68
7.6 Properties of the *ETFDM* . . . 69

8 The *DFDM* 71
8.1 FDs in the *DFDM* . . . 71
8.2 Creating a *DFDM* . . . 73
8.3 Handling aliasing with the *DFDM* . . . 74
8.4 Properties of the *DFDM* . . . 74

9 Modeling Expressions 77
9.1 Variable assignments . . . 77
9.2 Method calls . . . 79
9.2.1 Scope resolution . . . 80
9.2.2 Method resolution . . . 83
9.2.3 Getting the FDM of the called method . . . 89
9.2.4 Transforming a FDM . . . 90
9.2.5 Combining multiple external models . . . 93
9.2.6 Adding the method call model . . . 94
9.2.7 Comparing the *ETFDM* and *DFDM* . . . 94
9.3 Conditional expressions . . . 95

10 Modeling Statements 97
10.1 Modeling principles . . . 97
10.2 Blocks . . . 98
10.3 Selection statements . . . 99
10.3.1 Computing FDMs of selection statements . . . 99
10.3.2 Introducing self dependencies . . . 102
10.3.3 Other selection statements . . . 103
10.3.4 Model comparison . . . 104
10.4 Loop statements . . . 104
10.4.1 Modeling principles . . . 105
10.4.2 Computing the FDs of a **loop** . . . 107
10.4.3 Sub-traces for loop bodies . . . 114
10.4.4 Location creation in loops . . . 115
10.4.5 Modeling **do** and **for** statements . . . 118
10.4.6 Model comparison . . . 120

10.5 Synchronization statements 120
10.6 Try statements 121

11 Modeling Arrays & Strings **123**
11.1 `Java` arrays 123
11.2 Modeling arrays 124
11.3 An example *DFDM* 126
11.4 Modeling `Java` strings 127

12 Modeling Methods **129**
12.1 FD method models 129
12.2 Using default models 131
12.3 FD class, package, and host environment models 133
12.4 Modeling system classes 133

13 Handling Recursion **135**
13.1 Introduction 135
13.2 The method dependency graph 137
13.3 Top-level detection of infinite recursions 140
13.3.1 Polymorphism and recursion 141
13.3.2 Evaluation of expressions 141
13.4 Fix-point iteration modeling 142
13.4.1 Initialization iteration 143
13.4.2 Computing the next iteration 146
13.4.3 Termination of the fix-point algorithm 153

14 The *SFDM* **157**
14.1 Basics of the *SFDM* 157
14.2 Simplified functional dependencies 159
14.3 Creating a *SFDM* 160
14.4 Handling aliasing with the *SFDM* 161
14.5 Properties of the *SFDM* 163

III Debugging **167**

15 System Descriptions **169**
15.1 Diagnosis components 169
15.2 System descriptions 170

16 Computing Diagnoses **173**
16.1 Logical model description 173
16.2 Observations 174
16.3 Computing diagnoses 175

17 Building a Debugger **179**
17.1 The debugging process 179
17.2 Measurement selection 181
17.3 Variable query 183
17.4 Hierarchical debugging 184
17.5 Debugging method calls 185

18 Enhancements **187**
18.1 Assertions 187
18.1.1 Assertion syntax 187
18.1.2 Semantic restrictions 189
18.1.3 Using assertions for debugging 190
18.2 Using multiple test-cases 191
18.2.1 Extending MBD 191
18.2.2 Computing diagnoses using multiple test-cases 192

19 The JADE Debugging Environment **197**
19.1 The JADE system 197
19.2 The JADE debugger 198
19.3 An interactive debugging session 200
19.4 Hierarchical debugging 203

20 Empirical Results **209**
20.1 Evaluating the diagnosis performance 209
20.1.1 General remarks 210
20.1.2 Empirical diagnosis results 210
20.1.3 Discussion 216
20.2 Evaluating the debugging performance 219
20.2.1 General remarks 219
20.2.2 Empirical debugging results 222
20.2.3 Discussion 226

21 Discussion **229**
21.1 Diagnosis 229
21.1.1 Fault classes handled by the JADE debugger 229
21.1.2 Enhancing diagnosis using FDMs 231
21.1.3 Using alternative models 234
21.1.4 Outlook 235
21.2 Debugging 236
21.2.1 Enhancing the JADE debugger 236
21.2.2 An integrated software development tool 239

22 Conclusion **243**

List of Figures

3.1 Diagnosis process in MBD 24
3.2 Physical system 25
3.3 Combining MBD and Debugging 28
3.4 Diagnosis process in MBD 29

5.1 Example program *Point.java* 41
5.2 UML representation of class *Point* 42
5.3 The `Java` system $system^{5}_{test()}$ 45
5.4 Approximation of the `Java` system $system^{5}_{test()}$ 46

6.1 Example method *maxCoordinate()* 48
6.2 Example block b_1 56
6.3 Example block b_2 56
6.4 Internal model of block b_1 57
6.5 Internal model of block b_2 57
6.6 External model of block b_1 58
6.7 External model of block b_2 59

7.1 Example method *etfdm1(int x, int y)* 63
7.2 Evaluation trace of method *etfdm1(int x, int y)* 64
7.3 Example method *etfdm2(int x, int y, int z)* 65
7.4 Evaluation trace of method *etfdm2(int x, int y, int z)* 66
7.5 Example method *aliasing()* 69

9.1 Example method *assignment()* 78
9.2 Example class *obj* 80
9.3 Example class *obj2* 80
9.4 Example method *mc1(int i)* 81
9.5 Example method *mc2(int i)* 82
9.6 Example method *mc3(int i)* 84
9.7 Example method *mc4()* 86
9.8 Example method *mc5()* 86
9.9 Example method *mc6(int i)* 88

10.1 Example method *if1(int i)* 100
10.2 Example method *if2(int i)* 103
10.3 Example method *while1(int i)* 106
10.4 Transforming a loop statement into nested **if** statements 108
10.5 Digraph of the **while** body of Figure 10.3 111

10.6 Digraph of the **while** body of Figure 10.7 . . . 111
10.7 Example method *while2(int i)* . . . 112
10.8 Example method *while3(int i)* . . . 114
10.9 Example method *while4(int i)* . . . 115
10.10 Transforming a **for** loop into a **while** loop . . . 119

11.1 Example method *array1()* . . . 124
11.2 Example method *array2()* . . . 126

12.1 Variable environment of method *test()* . . . 131

13.1 Example interface *Queen* . . . 136
13.2 Example class *NullQueen* . . . 136
13.3 Example class *ConcreteQueen* . . . 137
13.4 Example class *EightQueens* . . . 138
13.5 MDG of class *ConcreteQueen* . . . 139
13.6 SCCs of class *ConcreteQueen* . . . 139
13.7 Locations created in iteration 0 . . . 145
13.8 Example method *rec1(int x)* . . . 145
13.9 Example method *rec2(int t)* . . . 149
13.10 Locations created in iteration 1 . . . 152
13.11 Locations created in iteration 2 . . . 153

14.1 Abstract view of the `Java` system $system^{5}_{test()}$ . . . 158
14.2 Example method *aliasing()* . . . 163
14.3 Example method *sfdm1()* . . . 165

15.1 System description of method *test()* . . . 172

17.1 The debugging process . . . 180
17.2 Measurement selection example 1 . . . 182
17.3 Measurement selection example 2 . . . 183

18.1 Example method *multipleTestcases1(int i, int j)* . . . 193
18.2 Example method *multipleTestcases2(int x, int y)* . . . 195

19.1 `JADE` modules . . . 198
19.2 The `JADE` debugger main window . . . 200
19.3 Specifying initial observations . . . 201
19.4 Selection and evaluation of a measurement point . . . 203
19.5 Debugging **if** statements . . . 204
19.6 Debugging **while** statements . . . 206
19.7 Stepping into an incorrect method . . . 207

21.1 Example method *demo()* . . . 232
21.2 System description of method *demo()* . . . 233
21.3 Example method *vbm()* . . . 234
21.4 System description of method *vbm()* . . . 235
21.5 Using different models during the debugging process . . . 240

List of Tables

2.1 Fault classes by their manifestations and effects 13
2.2 Sample bug statistics 14

20.1 Columns of Tables 20.2 to 20.7 211
20.2 Diagnosis results of test series *Adder* 212
20.3 Diagnosis results of test series *IfTest* 213
20.4 Diagnosis results of test series *WhileTest* 214
20.5 Diagnosis results of of series *Numeric* 215
20.6 Diagnosis results of test series *TrafficLight* 215
20.7 Diagnosis results of test series *Library* 216
20.8 Columns of Table 20.9 216
20.9 Average diagnosis results of all test series 217
20.10 Columns of Tables 20.11 to 20.16 222
20.11 Debugging results of test series *Adder* 223
20.12 Debugging results of test series *IfTest* 224
20.13 Debugging results of test series *WhileTest* 224
20.14 Debugging results of test series *Numeric* 225
20.15 Debugging results of test series *Trafficlight* 225
20.16 Debugging results of test series *Library* 226
20.17 Columns of Tables 20.18 and 20.19 226
20.18 Average total user interactions of all test series 227
20.19 Average variable query user interactions of all test series 228

Part I

Theoretical Foundations

Chapter 1

Introduction

Faults in software systems pose a serious problem to software engineers and users at the same time. Whereas users have to deal with an unexpected system behavior, for a software developer faults constitute unpredictable costs as far as both, money and development time, is concerned. Although there exists a variety of beliefs underestimating the crucial role of bugs in software projects, most of these beliefs do not hold in reality (see [2]). Therefore, efficient techniques and tools for an optimal detection, localization, and correction of software faults are important fields of research within the software engineering discipline.

Traditional debugging tools have been in use for the last couple of decades in order to exactly locate faults detected in one of the various test phases performed during the software development process. However, due to a lack of used information traditional debugging tools have only partly been able to serve their designated purpose. As a consequence, various approaches to build automatic debugging tools have been proposed. The majority of these approaches tries to improve the debugger's performance by the use of additional information and can therefore be named *intelligent debugging* techniques.

One approach to automatic software debugging is the application of techniques taken from model-based diagnosis. The model-based approach [38, 11] is based on the availability of a logical representation, i.e., a model, of at least the correct behavior of a technical system. By describing the structure of the system and the function of its components, it is possible to ask for sets of components, whose malfunction explains the detected misbehavior of the whole system. These sets can be seen as diagnoses of the system. Whereas model-based diagnosis has mainly been used to diagnose physical systems, its application to the software debugging domain has been proposed and tested on various occasions (see [8, 4, 5, 15, 43]).

The `Java` Diagnosis Experiments (`JADE`) project is a research project carried out by the Database and Artificial Intelligence Group of the Vienna University of Technology. The project has been funded by the Austrian Science Fund (FWF) under grant P12344-INF. It brings together the practical problem of creating an efficient debugging tool and the theoretically well-founded theories and algorithms of model-based diagnosis. Its goals are to extend the knowledge about the applicability of model-based techniques to the debugging of object-oriented

software systems and to implement a debugging prototype for the project's target programming language `Java`. Most practical descriptions and theoretical considerations discussed in this work are based on the `JADE` project. They are either direct results of the `JADE` project or have emerged during an intensive research of the abovementioned topics.

As every model-based approach is based on the existence of an appropriate model of the system to be analyzed, the main part of this work is dedicated to the creation of software models for debugging. In Part II of this work we show how different kinds of functional dependency models can automatically be derived from a given source code by making use of the underlying programming language semantics. In particular, we define the following three functional dependency model classes and discuss their respective properties:

- The Evaluation Trace Functional Dependency Model (*ETFDM*) computes all functional dependencies of a particular program run by making use of an evaluation trace. By doing so all data and control dependencies as they occur during run-time are known and can directly be incorporated into the resulting model. However, a single *ETFDM* is only valid for an individual program run, i.e., for a single test-case.

- The Detailed Functional Dependency Model (*DFDM*) is a static approximation of a model covering all functional dependencies as they might occur during run-time in all possible scenarios. Because the *DFDM* does not make use of evaluation traces, not all source code structures can be modeled using a purely static analysis. Therefore, as will be described in detail, the *DFDM* implies a higher level of abstraction than the *ETFDM*.

- The Simplified Functional Dependency Model (*SFDM*) is based on either the *ETFDM* or the *DFDM* and can be interpreted as a more abstract view of the underlying model. Whereas it is easier to read and understand due to its simpler structure, it is less exact and detailed than its underlying model.

The automatic creation of software models of a given source code system requires the transformation algorithm to be able to handle all source code structures defined by the target programming language. In particular, the system must be able to handle variable assignments, method calls, and conditional expressions at expression level, selection and loop statements at statement level, other source code structures, such as strings and arrays, and finally the transformation of whole methods and source code systems. Part II of this work discusses the transformation process of these source code structures in detail and highlights the potential advantages and drawbacks of the resulting model fragments.

The created models can then be applied to the debugging of `Java` programs. This is done by transforming the model into an internal logical system description, which together with a standard theorem prover and model-based diagnosis algorithms is used to compute diagnoses, i.e., bug candidates, for a faulty `Java` method. Part III of this work shows how our debugger prototype is constructed

building on the models and techniques described in previous chapters. In combination with modern GUI design such a tool can automatically highlight all potential bug positions in a `Java` method, which is known to be malfunctioning.

Furthermore, the debugging process can be improved by using a measurement selection algorithm, which automatically computes a particular point within the analyzed method. The evaluation of this point by the user eliminates an optimal number of incorrect diagnoses. In the software debugging case the user is asked to specify the value of a certain variable at a particular source code position, which helps the debugging tool to further reduce the number of bug candidates and thus more exactly focus the user on the parts of the program possibly containing the bug. We show how an interactive debugging tool is created, which aims at locating the exact source code position of a detected bug in a minimum of user interactions and thus in a maximal automatic fashion.

In the course of the `JADE` project an interactive debugging environment has been implemented and used to debug `Java` programs. In this work we briefly describe the architecture and functioning of the `JADE` system. We show the GUIs, which are needed in each step of the debugging process, and describe all user interactions performed by the system. Furthermore, we present two enhancements to the debugging process, which have both been incorporated into the `JADE` debugging environment. The first improvement deals with the concurrent application of multiple test-cases, which increases the debugger's diagnosis performance. The second enhancement is the design and implementation of an assertion language. This concept allows the user to specify observations in a clear way and further increases the debugging potential of the `JADE` tool.

Finally, we evaluate the performance of the `JADE` debugging prototype on all three underlying models. This is done by testing both, the debugger's diagnosis and debugging performance. Detailed results of all experiments with the debugging tool are presented and compared with each other. We show how well the debugger performs on various source code structures and discuss problems and weaknesses of the individual models. We also elaborate on different fault classes, which are covered by our approach and others, which are not. Building on the empirical results we present various possibilities of enhancing the performance of the `JADE` debugger. We conclude this work with a general discussion about the applicability of the `JADE` debugging tool to real-world debugging problems and the future role of model-based debugging approaches in the context of an efficient software development process.

More precisely, this work is organized as follows:

- Part I of this work deals with the principles of debugging and model-based diagnosis. Moreover, the `JADE` project is introduced. Part I includes the following chapters:
  - Chapter 2 discusses the technical and economic need for efficient debugging tools in the context of the software engineering process. It also gives a brief overview of existing (automatic) debugging strate-

gies. Furthermore, general thoughts about software faults and error classes are given.

  - Chapter 3 explains the basic concepts of model-based diagnosis and shows how they can be applied to the debugging of computer programs.
  - In Chapter 4 the `JADE` project is introduced, which aims at combining standard model-based techniques with traditional software engineering approaches by creating an debugging tool. An overview of the activities within the `JADE` project and the current project state is described.

- Part II of this work is dedicated to the modeling of `Java` programs for debugging. Format, properties, and creation of three different model types are discussed. Various examples are included in the text to highlight strengths and weaknesses of the individual models.

  - Chapter 5 discusses general properties of software models and `Java` systems. It shows how a `Java` system can be specified at compile-time and at run-time. Basic definitions for the following chapters are given. Furthermore, different model views are presented.
  - Chapter 6 deals with various forms and properties of functional dependency models in general. The different kinds of dependencies and modeling levels are discussed.
  - In Chapter 7 we introduce a first dependency-based model, i.e., the *ETFDM*, which makes use of run-time information in form of concrete evaluation traces. Its exact format and its creation are specified in detail.
  - Another dependency-based model is introduced in Chapter 8, i.e., the *DFDM*. In contrast to the *ETFDM* it is a purely static model, which is created without using any run-time information. Again, format and creation of the model are described in detail.
  - Chapter 9 is dedicated to the modeling of expressions. It shows how variable assignments, method calls, and conditional expressions can be incorporated into both models, the *ETFDM* and the *DFDM*.
  - In Chapter 10 we show how to model statements. The most detailed sections deal with the transformation of selection and loop statements, which constitute an essential part of any imperative and object-oriented computer program.
  - Chapter 11 is dedicated to the modeling of arrays and strings.
  - In Chapter 12 we explain how whole methods and `Java` systems can be modeled and how they can be represented in a concrete implementation. Further on, the use of default models and the modeling of system classes is discussed.
  - Chapter 13 is dedicated to the more complex issues arising from recursive programs. A fix-point algorithm is presented, which handles recursive method calls in a static analysis. A detailed example is given, which highlights the main properties of the fix-point modeling process.

  - Chapter 14 deals with a third model type, i.e., the *SFDM*. The *SFDM* is based on either the *ETFDM* or the *DFDM* and represents a simpler and more abstract model type than the underlying model. Format and creation of the *SFDM* are described.

- Part III of this work deals with the application of the models created in Part II to several debugging problems. The transformation of software models to logical system descriptions and their usage in a debugging tool is described. Further on, Part III contains empirical results obtained from concrete debugging experiments. Finally, a discussion about advantages and drawbacks of the used techniques and the quality of the obtained results rounds off this work. Part III contains the following chapters:

  - Chapter 15 describes how the models from Part II can be used to automatically create a system description of the analyzed `Java` system. This includes the creation of diagnosis components, which are then linked together by system connections to define the structure of the system to be diagnosed.
  - In Chapter 16 we show how a system description and observations of the system's behavior can be expressed in logical sentences. These sentences are then used together with a standard theorem prover and model-based diagnosis algorithms to compute diagnoses.
  - Chapter 17 is dedicated to the creation of a debugging tool making use of the models and techniques described in previous chapters. We describe the interactive process of locating the exact source code position of a certain bug by applying efficient measurement selection algorithms, variable queries and hierarchical debugging strategies.
  - In Chapter 18 we present two enhancements to the diagnosis performance of the `JADE` debugging environment. The concurrent use of multiple test-cases makes it possible to compute less diagnoses for a buggy `Java` method. The application of an assertion language supports the user in the efficient specification of observations. By incorporating assertions into the debugging process the debugging performance of the `JADE` tool is further improved.
  - The current version of the `JADE` debugging environment is briefly described in Chapter 19. We deal with the individual types of user interactions performed by the system during a debugging session and present the most important GUIs.
  - Chapter 20 deals with experiments carried out with the `JADE` prototype debugger and presents empirical results obtained from these tests. We discuss the diagnosis and debugging performance of the system in combination with all model types described in Part II of this work and analyze the strengths and weaknesses of the approaches presented herein.
  - Chapter 21 is dedicated to a detailed discussion about the approaches and empirical results presented in this work. We show, which fault

classes can be handled by the JADE debugging environment and which cannot, and present a variety of possible enhancements to the techniques described herein. Furthermore, we discuss the future role of model-based debugging systems in the context of an integrated software development tool.

– In Chapter 22 we conclude this work with a brief summary of the main issues raised in previous chapters. Finally, we list the main contributions of this work to the research area of model-based software debugging.

Chapter 2

Software Debugging

Errors, faults, and bugs in software systems seem to be clear and well-defined concepts, which on second sight turns out to be a misunderstanding in most cases. This chapter therefore starts with a discussion about the different concepts of software anomalies and shows in which forms and phases during the software development process these anomalies occur. We then motivate the need for efficient debugging tools, which possibly allow for an automatic fault localization. Finally, we discuss various approaches to automatic software debugging.

2.1 Terminology of software anomalies

When one speaks about finding an error or detecting a fault location in a software system, the terms *error* and *fault* are often used quite loosely. A first step in any work about testing, debugging, and automatic code correction has therefore to define the different meanings of these terms. In the IEEE Standard Glossary of Software Engineering [19] the following, more precise definitions, which stem primarily from the fault tolerance discipline, are given:

Mistake: *A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator.*

Fault: *An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program.*

Failure: *An incorrect result. For example, a computed result of 12 when the correct result is 10.*

Error: *The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.*

Mistake: As we see from the first definition, almost every misbehavior of a software system origins in an incorrect human action, i.e., a mistake. Assume

one wants to access the fifth element of an array, whose indices start with 0. Clearly, the mere intention to access the element with index five is a mistake. Unfortunately, the term *mistake* is not very useful in combination with well-defined technical problems, such as debugging or automatic testing. A wrong comment, thus, constitutes a mistake following the above definition, but definitely has no effect on the program's behavior.

Fault: Often a mistake results in a fault. In our example a fault is created by adding `anArray[5]` to the source code. When speaking about debugging, we are normally interested in exactly these program faults. Note that often the terms *error* and *bug* are used in a similar meaning.

Failure: Once a fault has been introduced into a software system, a failure is likely to occur. In our example a memory violation or the computation of an incorrect value might be the result of the abovementioned fault.

Error: An error in the strictest meaning of the word denotes the difference between a computed and a specified result. Assume that the fifth element of our example array stores the integer value 8. If we access `anArray[5]`, i.e., the sixth element, and get a value of 5, the difference between the expected and observed behavior represents an error. In this case the error can be defined as the arithmetic value of 3.

It should be mentioned that there exists a variety of further terms for describing the anomalies of software systems. Some of these are briefly discussed in the following list. However, it should be noted that in this work we try to adhere to the IEEE definitions given above.

Anomaly: In [19] the term anomaly is defined as *anything observed in the documentation or operation of software that deviates from expectations based on previously verified software products or reference documentation.* The definition given in the IEEE Standard Classification for Software Anomalies [18] expands upon this definition by including deviations from the user's perception or experiences. Therefore, *anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation.* It is easy to spot that in both definitions the term *anomaly* is used very widely and comprises the concepts of errors, faults, and failures.

Defect: Some works (e.g., [7]) distinguish between the static component of a fault, i.e., a defect, and its dynamic component, i.e., a fault. By using these definitions a defect is an incorrect piece of source code (e.g., an incorrect assignment statement), whereas a fault can be seen as an improper or unacceptable software state resulting from a defect. Note that this distinction does not conform to the IEEE definitions as defined above. Therefore, we herein forgo to make this distinction and only use the term *fault.*

As mentioned above, in this work we use the IEEE standard definitions for the terms *fault*, *failure*, and *error* as described above. The term *bug* is used in the same meaning as the term *fault*. Other terms are avoided as much as possible in order to make this work more precise. In the following sections we will focus on software faults as they appear in a system's source code.

2.2 Classifying software faults

In scientific literature there exists a variety of different criteria, which can be applied to the classification of software faults. Some of these criteria are (taken from [7]):

Software constituent: Faults can be found in programs, data structures, and documents. In this work we mainly focus on the debugging of programs, i.e., the detection of program faults (bugs). This, however, should in no way underestimate the problems and costs associated with incorrect data structures and documentations.

Life cycle: Faults can be introduced into a software system during any phase of its life cycle. We can therefore classify faults by their accruement, e.g., requirement, design, coding, testing, etc...

Manifestation: Faults can manifest themselves as purely textual faults (e.g., spelling faults) or conceptual faults (incorrect method calls). Further on, faults may be classified into grammar faults or faults in the semantics of the respective target language.

Cause: Faults can be classified according to the nature of the human mistake, which originally introduced the fault. There are multiple reasons for the existence of software faults, such as technical, organizational, historical, group dynamic, individual, and other reasons (see [7]).

Consequence: Faults can be classified according to the nature of failures which are the fault's consequence.

As an example classification of software faults by their consequences we define the following classes. Note that following the IEEE definitions given in Section 2.1 these classes denote failures or error classes, rather than fault classes.

Compile-time errors: A certain fault may result in a compile-time error. Usually, these faults can be located quite efficiently through the use of compiler error messages and warnings.

Run-time errors: Dynamically, a certain fault may cause a run-time error. These faults can sometimes be located very easily (e.g., in the case of proper exception handling), but in the general case pose a difficult problem for a debugger, e.g., memory violations and core dumps.

Output errors: If a program does not terminate abruptly, i.e., with a run-time error, there is still the possibility to obtain an incorrect result, i.e., an error. This observable error can then be used to locate the exact position of a fault in the source code. Note that herein we focus mainly on locating faults resulting in an observable output error.

No output errors: Finally, a certain fault may not cause an error for a particular test-case at all. In this case debugging is very hard, especially as there seems to be no need for fault locating. In the following chapters we rely on a successful test-case, i.e., on the fact that a fault has already been detected. Source code faults, which stay undetected during testing, are not discussed herein.

As an example classification of software faults by their manifestation we define the following fault classes:

Syntax errors are software faults, which are not consistent with the specified grammar of a certain computer language. These faults are always detected at compile-time.

Semantic errors are faults in contradiction to a language's semantics. Whereas *errors of the static semantics*, e.g., incorrect types, are detected at compile-time, *errors of the dynamic semantics* generally cause run-time errors, e.g., division by zero or (in some programming languages) method calls on objects, whose class does not implement the method.

Logical errors are software faults resulting from logical mistakes made by the programmer, e.g., wrong data structures, use of an incorrect variable, etc... These faults might cause an observable output error, but may as well stay undetected for a certain program input. Once logical errors become observable, they constitute the primary target of most debuggers.

Note that the above anomalies are all called *errors*. Strictly speaking, they are failures and faults, respectively, following the definitions in Section 2.1. Table 2.1 shows both classifications (by the manifestation and effect of a particular fault, respectively) and highlights the relationship between these two classifications. As already mentioned in this work we focus on source code bugs, which manifest themselves as logical faults resulting in an output error. In order to further divide this class into sub-classes, which are used in the following sections of this work, we define:

Functional faults are source code faults, which result in a certain variable storing an incorrect value in at least one possible evaluation trace. Nevertheless, functional faults do not alter the structure of the program, which means that the dependency graph [14] of the buggy program is equivalent to the dependency graph of the correct program. Examples of these faults are the specification of incorrect operators or incorrect literals.

Manifestation	Effect			
	compile-time	run-time	error	no error
Syntax error	×			
Semantic error (stat.)	×			
Semantic error (dynam.)		×		×
Logical error			×	×

Table 2.1: Fault classes by their manifestations and effects

Structural faults are source code bugs, which alter the dependency graph of the program. Examples of this fault class are the access of an incorrect variable or missing and superfluous source code structures.

The basic fault classification schemes introduced in this chapter are quite coarse and not specific to certain computer languages. In the following section we present a more detailed fault taxonomy together with some fault statistics.

2.3 Fault taxonomies & statistics

Standard fault taxonomies can mostly be found in scientific literature about software engineering and software testing. Generally, they are used to keep track of all faults appearing in a particular software system during all phases of the system's software life cycle. The main reason for such classifications is to gain as much knowledge about potential bugs as possible in order to make the software development process more efficient and thus less expensive. To give an impression of the plurality and heterogeneity of software faults the following list of faults (taken from [2]) shows a sample fault taxonomy. Table 2.2 shows a breakdown of software bugs. A more detailed breakdown can be found in [2].

Requirements and specification: This class includes all faults in system requirements and specifications, such as incomplete, ambiguous, or (self) contradicting documents. Clearly, these faults pose a serious problem for software engineers, especially if they appear early in the software development process and are only detected in late phases of the software life cycle.

Features and Functionality: This class includes all wrong, missing, and superfluous features and functionalities of a given system. Whereas missing features are normally easy to detect, superfluous functionality is more problematic as it increases the probability of faults in following phases.

Structural bugs: This class can be further divided into:

- Control flow and sequence faults, e.g., paths left out, improper nesting of loops, etc...
- Logic fault, e.g., wrong use of **switch** statements or logical operators, etc...

Fault class	Total	%
Requirements	1,317	8.1%
Features and functionality	2,624	16.2%
Structural bugs	4,082	25.2%
Data	3,638	22.4%
Implementation and coding	1,601	9.9%
Integration	1,455	9.0%
System, software architecture	282	1.7%
Test definition and execution	447	2.8%
Other, unspecified	763	4.7%

Table 2.2: Sample bug statistics

- Processing faults, e.g., arithmetic bugs, algorithm selection, general processing, etc...
- Initialization bugs, e.g., improper or superfluous initialization, etc...

Note that this definition is not equivalent with the definition of structural faults given in Section 2.2, which we are using throughout this work.

Data: This fault class comprises all bugs arising from incorrect data objects, their numbers, initial values, etc...

Coding bugs: typographical bugs, misunderstanding of the operation of a particular statement, documentation errors.

Interface and integration faults: interfaces to other systems, hardware, etc...

System faults: These faults result from the interaction between many components, such as programs, data, hardware, operating system.

Test: This class includes all faults arising during the test phase, i.e., incorrect test-cases, incorrect testing algorithms, etc...

[2] also gives a brief overview of some fault statistics. The tested program had a total of 6,877,000 statements including comments. The total amount of bugs reported was 16,209, which on average amounts to 2.36 faults per 1000 statements. Table 2.2 shows a detailed breakdown of the number of reported faults and the percental share in all reported faults for each class.

As mentioned above a fault classification scheme as presented above can be very important for a company's software engineering process in the way that it increases its efficiency and helps reducing costs. Furthermore, it is a handy concept for demonstrating the plurality and variety of existing faults in software systems. On the other hand, fault classification schemes seem to be of little use for a theoretical task, such as measuring the performance of a debugger. This is because of the incompleteness and ambiguity of these schemes. For instance, the distinction between certain structural bugs and data faults does not seem to be

clear. Given a particular fault, there seem to be more than one possible classes, which can be assigned to this fault.

More interesting from the debugging point of view is the statistical analysis of concrete faults presented above. As the different fault classes need different testing and debugging techniques, the individual percentages tell us how many faults we will likely be able to handle with a certain testing or debugging technique. In the following we briefly discuss the individual fault classes in the context of debugging potentials:

Structural bugs (25.2%): Together with data faults this class represents the largest fault class. Locating faults belonging to this class is the primary goal of traditional debugging tools, e.g., detecting the execution of incorrect paths, the computation of incorrect values, etc... Therefore, the goal of this work is to locate structural bugs in the first place. It should again be noted that in the following chapters we use the definition of structural faults as given in Section 2.2 instead of the more general definition used by [2].

Data faults (22.4%): Although often not considered as faults, data faults are as unpleasant and almost as frequent as structural bugs. Incorrect data objects, faulty formats, and buggy initial values should be located by any debugging tool. Thus they are within the scope of this work, too.

Features & functionality (16.2%): Missing or incorrect features of a certain software system represent a large class of software faults. Often they are introduced by problems in human-to-human communication and require very specific solutions, such as high-level, formal specification languages. Both, traditional debuggers and the debugger constructed herein are not designed to locate and correct feature bugs.

Implementation & coding (9.9%): Coding faults, although quite frequent, are not within the scope of this work. Whereas we assume that purely typographical faults are detected and repaired by the programmer at compile-time, we are not interested in documentation faults or violations of style conventions and programming standards. This, however, should not deny the importance of proper documentations and the adherence to software standards, especially as these concepts help reducing the amount of faults during maintenance phases.

Integration (9.0%): Integration faults seem to be an important fault class, too. As far as all interfaces between certain system components are clearly defined and testable, integration faults are a common target of traditional debugging approaches.

Requirement faults (8.1%): The class of faults in requirements is quite large and therefore not to be underestimated. Nevertheless, these faults require special testing and debugging techniques, which are not part of this work.

Test definition & execution (2.8%): In this work we assume that all test specifications and test-cases are correct. We therefore ignore the possibility of such faults.

System, software architecture (1.7%): Here the same is true as for the test definition and execution fault class. It is assumed that no problems with the hardware system, operating system, etc... occur.

Therefore, this work focuses on structural, data, and parts of the integration faults of a given software system. If all these faults could be found, over 50% of all software anomalies appearing throughout the software life cycle could be handled more efficiently. The economic gain of such a system seems to be evident. All other faults (mainly feature, coding, and requirement faults) are either easy to locate, e.g., typographical faults, or require special-purpose tools and techniques. The location of these faults is beyond the scope of this work.

2.4 Software debugging

As Section 2.3 indicates, software faults appear in almost every software system from medium-size stand-alone applications to large distributed software systems. Clearly, these faults pose a lot of problems to the users and software engineers, which can be summarized as follows:

- From a user's point of view faulty software systems produce an incorrect output for at least some input combinations. This seems to be especially problematic, if we talk about security critical systems, e.g., airplanes or nuclear plant controllers, which not only affect a small number of users, but large parts of the population.
- From a software engineering point of view, software faults can cause enormous costs. Interestingly, the cost of a software fault increases in later stages of the software development process, i.e., the later a certain bug is detected and corrected the higher the costs for the software engineer.

As a consequence, it has to be one of the primary goals of every software engineer to either produce fault free software in the first place, or to detect, locate, and correct an existing bug as soon as possible in order to reduce the overall development and maintenance costs. In practice it showed that the first approach is a very limited one. Although there exists a variety of techniques and tools, which help to avoid the introduction of software faults in the first place (e.g., automatic software generation, formal verification, programming standards, etc...), bugs can still be found in almost all software systems. An efficient localization and correction of these faults is therefore one of the most important stages during the whole software development project. This is exactly the point where software debugging (SD) enters the scene. In a broad definition SD comprises the detection, localization, and correction of software faults. The IEEE give the following definition in their Standard Glossary of Software Engineering [19]:

Definition 2.4.1 (Debug) *To detect, locate, and correct faults in a computer program. Techniques include use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operation, and traces.*

In a more narrow definition SD can be seen as the process, which takes place after a certain failure or error has been detected, i.e., after the execution of a successful test-case. Using this definition (see [33]) SD only includes two phases, i.e., (1) fault localization and (2) fault correction. Regardless of the exact definition and software engineering paradigms the following three steps have to be performed during the course of all software development processes:

Fault detection: As already mentioned, software faults should be detected and fixed as soon in the software development process as possible. Whereas earlier software engineering concepts relied on test phases at the end of the development process, other concepts are based on a continuous evaluation and testing of the current system. The main technique for detecting faults is software testing (see [33, 2]), but there exist other approaches like formal verification. The detection of software faults is out of the scope of this work. In the following chapters we rely on the fact that a misbehavior of the system has already been detected in form of a successful test-case.

Fault localization: Once a misbehavior of the system under consideration is detected, the fault should be located as soon and precisely as possible. It can be said that fault localization is the key problem in an efficient debugging strategy. Therefore, the efficient localization of software faults is the primary goal in this work. Note that when talking about debugging in most cases we mean the exact localization of a detected software bug.

Fault correction: Finally, a located fault has to be corrected in order to adapt the system's behavior to its specification. Interestingly, this does not necessarily have to be a trivial task, which can be demonstrated with the fact that quite often new software faults are introduced during the correction of an existing bug. Nevertheless, the main problem in SD is the fault localization, which accounts for most of the time and thus costs during the debugging process. In this work we do not deal with fault correction techniques (see [43]).

Following the definitions in Section 2.1, the process of testing and debugging can be described as follows:

1. Representative test-cases are created in to test the correctness of a particular piece of code. If there exists no explicit knowledge about possible mistakes made by the programmer or faults in the source code, the goal of testing is to produce failures or errors.

2. When running the tested piece of code on the specified test-cases, all errors are observed.

3. In a next step one tries to find faults, starting with the observed errors.

4. Once a fault is located it is corrected, hopefully without making any mistakes.

Therefore, the goal of software debugging is to locate faults in a program's source code. Errors and failures have to be seen as external effects of these faults, which in contrast to faults are directly observable. Only via the observation of errors and failures software faults can be found. Interestingly, SD is one of the most difficult part of the software development process, which most of the software developers dislike. The following reasons for this are given in [33]:

- Programmers often tend to rule out the possibility of making mistakes during the design and implementation of a certain software project.
- SD seems to be the most mentally challenging activity of all software development activities. In many cases during debugging a software engineer faces a high amount of organizational and self induced pressure.
- SD itself seems to be an inherently complex task. This becomes clear, if we look at the fact that generally the location of a given bug can be potentially any statement or expression of the program. Unlike physical systems an early ruling out of certain sub-systems is in most cases not possible.
- Compared to all other software development activities, comparatively little research, literature, and formal instruction exists on the process of SD.

In the end debugging seems to be one of the most important software engineering disciplines and the most underestimated and unpopular activity as far as both, programmers and researchers, are concerned. This work is therefore dedicated to (1) extending our knowledge about software debugging, (2) making the debugging process easier and cheaper for software engineers, and (3) guaranteeing a higher standard of the resulting software products for all users.

2.5 Debugging techniques

Traditional debugging techniques are mainly based on the idea of stepping through code, which is known to contain at least one buggy statement or expression, step by step and manually monitoring the current variable environment until any deviations from its expected state can be observed. Clearly, this is not a very efficient approach, because (1) it does not make use of any information other than the current evaluation trace and (2) can be seen as a random (extensive) search in the universe of all potential bug candidates, which in the worst case finds the bug only in its very last step. Additional techniques have been proposed and used in practice to overcome these drawbacks and make the debugging process more efficient. Among others, breakpoints, dumps, reversible execution, and assertions have been used together with powerful visualizing tools supporting the user in locating source code faults. Nevertheless, traditional debugging tools still seem to be far from being effective and satisfying in practice.

This is why over the years automatic software debugging has become a lively area of research and a wide variety of different approaches and systems have emerged. [13] gives an overview of some of the existing automatic debugging approaches and divides them into the following three categories:

Verification with respect to specifications is based on the idea of comparing a program with an existing formal specification. All parts of the program, which are not equivalent to the given specification, have to be deemed as suspect. However, a number of difficulties come with this approach: (1) Complete and accurate formal specifications are needed. The creation of these specifications is a difficult and costly task, which in many cases is simply impossible. (2) Any detected inconsistencies can be due to both, faults in the source code or faults in the specification. Therefore, both systems have to be checked in the case of deviations. (3) Specifications and implementations might be so different that a convincing comparison is not possible. In this case the whole part of the program covered by the specification might look suspect.

Checking with respect to language knowledge systematically parses programs and searches for language dependent faults. If a language structure does not conform to a positive rule or conforms to a negative rule, the structure is seen to be suspect. This is a very powerful tool, which eliminates certain faults very efficiently. The problem is that certain fault classes cannot be detected by a system, which only relies on knowledge about the programming language.

Filtering with respect to a symptom successively reduces the search space by filtering all source code structures, which cannot have produced the given fault symptom. This technique relies on the existence of an appropriate filtering criterion, which must not move a bug out of the search space.

In the following we concentrate on the last category of fault localization approaches, i.e., on filtering techniques. As already mentioned, these techniques remove parts of the source code, which can be proven not to account for a given fault, from the debugging scope and so try to narrow the remaining search space as much as possible. The following approaches can be seen as example filtering techniques:

Algorithmic Debugging: The idea of a software tool, which automatically locates and repairs software faults, was originally proposed by Shapiro in his dissertation [39]. Shapiro's approach is based on locating faults at the level of method calls, which is done by distinguishing between several algorithms handling incorrect and missing method calls separately. [39] also shows how a bug can be repaired once its exact location has been detected. However, his approach has several drawbacks, such as a high amount of required user interaction and the fact that the proposed algorithms cannot easily be applied to a large class of programming languages.

Program Slicing: (see [48, 49, 46]) computes a subset of a given program, which produces the same output as the original program for a set of variables at a certain position in the source code. For example, a slice on the slicing criterion $\langle 10, \{x\}\rangle$ contains all statements, which are needed to compute the value of variable x in statement line 10. This can be achieved by deleting all

statements, which are not needed for the computation of the values of the specified variables. Program slicing can be used to limit the search space by only looking at the slices of the variables, whose values are known to be incorrect.

Probabilistic Debugging: [6] introduces the notion of probability values into automatic software debugging. The approach is based on (1) computing all potential fault locations and (2) determining the statements, which most likely include the fault by using a belief network (Bayesian Network). However, this approach relies on the existence of a priori fault probabilities of different types of statements and expressions. It is thus doubtful, whether satisfying results can be obtained in the general case.

Model-Based Diagnosis: (MBD) [38, 11] makes use of a logical representation of the software system under consideration. This representation describes the whole system as a set of interconnected components, which may or may not account for a given error. By using information about the programming language semantics and possibly additional information one can make statements about which components might contain the bug and which do not have to be considered for further debugging. So far several authors have proposed the use of MBD for software debugging (see [8, 4, 5, 15, 43]).

Another field of research within the automated debugging community is the creation of efficient automatic tutoring tools. The main differences between software debugging of real-world applications and program tutoring are given in the following list. Note that in this work we focus on the task of locating bugs in real-world applications. Topics arising in the context of tutoring, training of novice programmers, or E-learning are not within the scope of this work.

- Whereas tutoring normally aims at small, simple, and well-known toy programs, general debugging environments are faced with sometimes large, very complex, and unique applications.
- In case of tutoring, a reference implementation either exists or can easily be generated. This is by no means the case in general debugging projects.
- The amounts and types of bugs vary between novice programmers and experienced programmers. Whereas in the general case it can be assumed that a correct program can be obtained by slightly altering the buggy source code, the tutoring task has to deal with program parts and data structures, which are completely incorrect and cannot be corrected by simple modifications.
- Tutoring systems are normally used to help novice programmers developing their first programs. In a general debugging scenario, on the other hand, a professional software developer faces a mostly new, real-world debugging problem under a high organizational pressure and strict time constraints.
- Failures in tutoring systems are mostly detected by looking at a reference solution. In the general case software testing is performed in order to evaluate the performance of a given software system.

In the following chapter we describe the principles of model-based diagnosis in more detail and show how this approach can be used for software debugging.

Chapter 3

Model-Based Diagnosis & Debugging

Model-based diagnosis (MBD) is a well-known AI technique for the localization of malfunctioning parts in (mostly physical) systems. In this section we briefly recall the basic definitions of MBD as given by [38] and show how this approach can be used to locate faulty components in a given system. We then discuss how MBD can be applied to the software domain by using standard MBD techniques to debug computer programs. The resulting approach of model-based software debugging (MBSD) serves as a basis for the debugging of `Java` systems, which is described in the following chapters of this work.

3.1 Model-based diagnosis

As mentioned above, the goal of MBD [38, 11] is to efficiently locate faulty components of a given technical system. The basic idea behind MBD is to have a logical representation of at least the correct behavior of the analyzed system, i.e., a model of the system, and a set of observations of the system's behavior. The used model must consist of components, which individually might be responsible for a misbehavior of the whole system. Such a misbehavior is detected whenever the observed behavior contradicts the behavior that is derived directly from the model. Figure 3.1 shows the basic process of diagnosing a given system using MBD. A set of inputs to the system and the system description are used to derive the expected (correct) behavior of the system, i.e., the resulting correct outputs. If the system to be diagnosed is not working correctly, the outputs produced by it do not agree with the expected values. The task of the diagnosis engine is to conclude, from the discrepancies between expected and observed values, which components in the system must have malfunctioned to produce the observed outputs.

In the last decade MBD has achieved wide recognition in the diagnosis community and has been applied to a wide variety of diagnosis problems, mainly the fault localization in technical systems, e.g., digital circuits (see [9, 10]) or power transmission networks (see [3]). This is due to the following advantages of MBD:

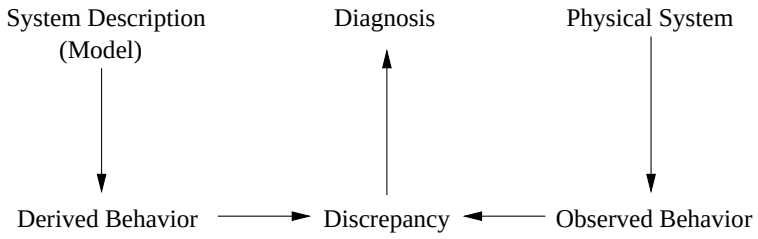


Figure 3.1: Diagnosis process in MBD

- Diagnosis is performed without explicit knowledge about how to locate faulty components. As a consequence, MBD can be seen as a domain independent technique.
- The used model only has to represent the correct behavior of components and the correct behavior of systems. A priori knowledge about the incorrect behavior of a given system and fault modes of its components can but does not have to be incorporated into the model.
- Once an adequate model has been developed for a particular domain, it can be used to diagnose different systems of that domain without changing the underlying modeling principles.
- The model can be used to search for single or multiple faults without alteration.
- The model does not depend on the underlying diagnosis algorithms and vice versa. This means that different diagnosis algorithms can be used for a given model and multiple models can be used with the same diagnosis algorithm.
- The existence of a clear formal basis for judging and computing diagnoses.

Let us now briefly recapitulate the basic definitions of model-based diagnosis as given by [38]. In the following SD is a logical model describing the (correct) behavior of a system, i.e., the system description, $COMP$ a set of components, and OBS a set of observations. We further assume SD and OBS to be sentences in first-order-logic. The system description makes use of the predicate $AB(C)$ ($\neg AB(C)$) to specify the incorrectness (correctness) of a component $C \in COMP$. The term $AB(C)$ ($\neg AB(C)$) says that component C behaves abnormally (normally). More formally, we define:

Definition 3.1.1 *A diagnosis system is a pair (SD, COMP) where*

- *SD, the system description, is a set of first-order sentences;*
- *$COMP$, the system components, is a finite set of constants.*

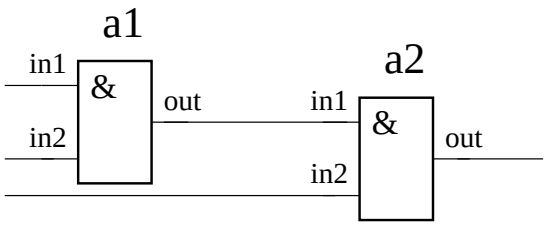


Figure 3.2: Physical system

Definition 3.1.2 *An observation of a system is a finite set of first-order sentences. The triple (SD, COMP, OBS) is called a diagnosis problem for the system (SD,COMP) with observations OBS.*

Example 3.1.1 *Consider, for instance, an integrated circuit, which consists of logical* `AND` *gates as its components (see Figure 3.2). We can create a logical model (system description) by specifying the* `AND` *gate behavior and a set of formulae that describe the interconnections of the components, i.e., of the gates in the circuit. This could read as follows:*

$$and\_gate(x) \wedge \neg ab(x) \supset out(x) = and(in1(x), in2(x))$$

$$and\_gate(a1) \ \ \& \ \ and\_gate(a2) \ \ \& \ \ out(a1) = in1(a2)$$

If we now specify observations of the system, we get a diagnosis problem. Consider, for instance, the following set of observations, which describe the behavior of the system depicted in Figure 3.2:

$$in1(a1) = 1$$

$$in2(a1) = 1$$

$$in2(a2) = 0$$

$$out(a2) = 1$$

If we take a certain diagnosis problem, i.e., a diagnosis system $(SD, COMPS)$ together with a set of observations OBS, there are two cases to be considered:

- $SD \cup \{\neg AB(C) | C \in COMP\} \cup OBS$ is consistent: in this case no malfunction of the system can be observed with the set of observations OBS. Note that this does not necessarily mean that all components of the system exhibit a correct behavior.

- $SD \cup \{\neg AB(C) | C \in COMP\} \cup OBS$ is inconsistent: in this case there exists at least one component in the system, whose behavior differs from its expected behavior, i.e., $\exists C \in COMP \mid AB(C)$. Of course, the number of incorrect components is not limited to just one component. If a malfunction of the system is spotted, we are interested in all components, whose malfunction explains the incorrect behavior of the whole system. In other

words, it is the goal of the diagnosis process to compute all sets of components, whose malfunction explains the incorrect behavior of the system.

Definition 3.1.3 *A diagnosis for (SD, COMP, OBS) is a set* $\Delta \subseteq COMP$ *such that* $SD \cup OBS \cup \{AB(C) \mid C \in \Delta\} \cup \{\neg AB(C) \mid C \in COMP \setminus \Delta\}$ *is consistent.*

A diagnosis is said to be minimal if no proper subset is itself a diagnosis. From Definition 3.1.3 it follows that every superset of a diagnosis is also a diagnosis. In practice, one is generally interested in finding minimal diagnoses, i.e., a minimal set of components whose malfunction explains the misbehavior of the system. Otherwise, one could explain every error by simply assuming every component to be malfunctioning.

Example 3.1.2 *Let us come back to the above diagnosis problem for the integrated circuit depicted in Figure 3.2. The assumption that all components behave correctly leads to a contradiction, since then, according to the behavior model, the correct value of* $out(a2)$ *would be 0. Therefore, some non-empty set of components must exist whose malfunction explains the misbehavior. In this case,* $AB(a2)$ *is the only minimal diagnosis, since assuming that* $AB(a1)$ *and* $\neg AB(a2)$ *still results in a contradiction.*

The dual concept of a diagnosis, which is used for computing diagnoses, is a conflict. A conflict specifies a set of components, which given the model and observations cannot all work correctly at the same time. In other words, a conflict always contains at least one component, which does not exhibit the expected behavior. More formally, we write:

Definition 3.1.4 *A conflict set for (SD, COMP, OBS) is a set* $CO \subseteq COMP$ *such that* $SD \cup OBS \cup \{\neg AB(C) \mid C \in CO\}$ *is contradictory.*

Example 3.1.3 *Looking at our integrated circuit example (see Figure 3.2), we find two conflicts. Whereas the set* $\{a1, a2\}$ *is a trivial solution,* $\{a2\}$ *constitutes a conflict on its own, because* $AB(a2)$ *is part of all possible explanations of the incorrect behavior of the whole system.*

Generally, the computation of diagnoses from conflicts makes use of the concept of hitting sets. Formally, we write

Definition 3.1.5 (Hitting Set, Reiter [38]) *Let* C *be a collection of sets. A hitting set for* C *is a set* $H \subseteq \bigcup_{S \in C} S$ *such that* $H \cap S \neq \emptyset$ *for each* $S \in C$*. A hitting set is minimal if no proper subset of it is a hitting set.*

Example 3.1.4 *For example, all minimal hitting sets for* $\{\{1, 2\}, \{1, 4\}\}$ *are* $\{1\}$ *and* $\{2, 4\}$*.* $\{1, 2, 4\}$ *is also a hitting set but it is not minimal.*

In [38] Reiter introduces the hitting set algorithm for computing diagnoses using a set of conflicts. This algorithm was improved by [17]. The relationship between diagnoses and conflicts is stated by the following theorem:

Theorem 3.1.1 (Reiter [38]) *The set $\Delta \subseteq COMP$ is a (minimal) diagnosis for $(SD, COMP, OBS)$ iff Δ is a (minimal) hitting set for the collection of conflict sets.*

3.2 Model-based debugging

Whereas MBD research mainly focuses on diagnosing physical systems, several authors [8, 4, 5, 15, 43] have proposed the use of model-based techniques in software debugging. Console et al. [8] introduce a model for debugging Prolog-like languages. The authors claim that their approach improves Shapiro's algorithmic debugging [39] by reducing the required user interaction necessary for locating a bug. Bond et al. [4, 5] critically analyze the work done by Console et al. [8] and show that the exception form for diagnoses is not canonical, leading to an incomplete diagnosis computation procedure. To overcome this problem Bond et al. propose an improved algorithm for debugging which also generalizes the declarative error diagnosis approach from Shapiro [39]. Friedrich et al. [15] introduce a system for debugging hardware designs written in the hardware description language VHDL. The authors use a dependency-based model for debugging. Because of the simplicity of the model a prototype implementation is able to debug even very large programs. Stumptner and Wotawa [43] discuss the use of model-based diagnosis in debugging more theoretically. The debugging of functional programs using MBD techniques was proposed in [42, 43] and eventually the debugging of object oriented systems was tackled by [44] (`Java`) and [37] (C++). In this section we how the standard MBD approach can be applied to the debugging of computer programs in general. Parts II and III of this work deal with modeling and debugging of `Java` systems.

The basic idea behind model-based software debugging (MBSD) is to derive a model directly from the program and the programming language semantics. This model has to distinguish components, describe their behavior, and the structure of the program under examination. The principles of MBSD are depicted in Figure 3.3. The program, in our case written in `Java`, is compiled into an internal representation. From this representation (together with a set of *model fragments*) a converter computes logical models for diagnosis. Model fragments represent a logical description of parts of a model, e.g., the behavior description of functions. This knowledge has to be derived from the programming language semantics. A model of `Java` programs, for instance, requires the model fragments of all basic functions and types of statements of the `Java` programming language. For example, the behavior of the "+" operator must be defined as $\neg AB(C) \rightarrow out(C) = in_1(C) + in_2(C)$.

After building the model, which is done automatically, the model together with the specified behavior of the program, e.g., test-cases, is used by the diagnosis engine to find bug candidates. The candidates can be further discriminated by adding additional knowledge, i.e., values of variables at specific points within the program. The selection of the variable and location is done by a measurement selection algorithm. The information about the value must be delivered by the user (or another oracle). The remaining candidates provide a link back to the

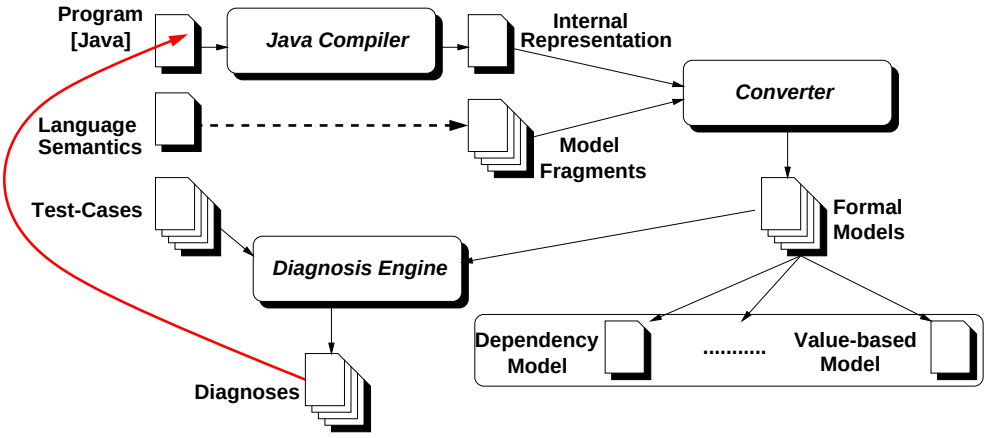


Figure 3.3: Combining MBD and Debugging

original source code.

When applying MBD to the debugging of computer programs, the basic approach remains the same, but a number of differences have to be taken into account. Compare Figures 3.1 and 3.4 to see the most notable changes during the diagnosis process. In the following some properties of diagnosis systems are discussed in the light of software debugging:

System description: The system description, as defined by Reiter, defines the structure and behavior of the system. Since the system to be diagnosed is a program, the system description is a description of the behavior of the program, which is derived from the program and the semantics of the language in which the program is written. Therefore, the model chosen usually varies with the language involved, because different languages have different semantics. Given a fixed choice of how the semantics should be represented, the system description can be derived automatically from the code of the program. Note that unlike in many hardware oriented applications the system description mirrors the bugs in the program. It is not an independent, correct specification, but rather includes the representation of the bugs the program contains in the first place.

Components: In model-based debugging the choice of components depends on the desired level of abstraction chosen for the used model of the source code. Whereas a very abstract model might not cover enough information for an efficient debugging process, a too detailed model might result in a very slow diagnosis process, which is of no practical use. As we will see in Chapter 5 in the case of `Java` systems there exist multiple levels of abstraction, which make sense for a representation of a `Java` program. The most important levels are the expression and statement level, which associate each expression (statement) of the system with exactly one component. Note that once the choice of model has been made (which parts of the program are represented as components and which parts of the semantics of the language are

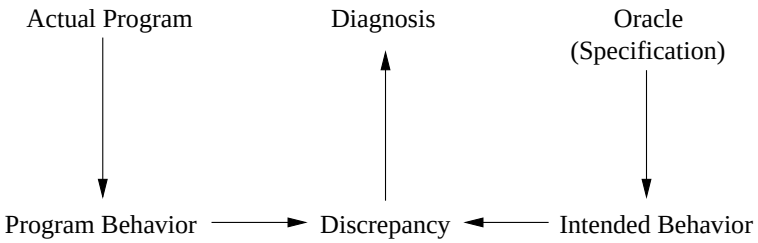


Figure 3.4: Diagnosis process in MBD

to be represented), converting a program to a model for a diagnosis system is straightforward and technically trivial, because the program must be available in machine-readable form in order to be executed.

Observations: In hardware diagnosis observations describe the behavior of the system. If a system is faulty, the observations therefore differ from the correct behavior predicted by the system description. In debugging, the roles have been reversed. It is the system description (derived from the buggy source code) that reflects the incorrectness of the program and whose output (incorrect in places) is confronted with observations that are correct (the correct output to be produced by the test data). Note that in traditional diagnosis problems the model is correct and it is the observations, made from the behavior of the system, that reflect the incorrect behavior. In addition, the question of how a programming faults may manifest itself in the model leads us to the related issue of structural faults (see Section 2.2).

Structure: Structural faults are faults that are not caused by an incorrectly functioning component, but by a missing or *additional* connection between two components, as in a bridge fault in electrical engineering. Structural bugs are very relevant in the field of software debugging, since many typical bugs are structural in nature. The use of an incorrect argument in an expression (e.g., by using a different variable name, switching the order of arguments), or the omission of part of a complex expression constitute typical examples of these faults. The usual way of dealing with structural faults is to assume the existence of a different, complementary model that allows to reason about the likelihood of such faults. In software, such models could take the shape of considering name misspellings, variable switchings, or attempts to repair expressions, i.e., synthesize missing parts, to provide correct functionality.

Chapter 4

The JADE Project

The Java Diagnosis Experiments (JADE) project[1] is a research project carried out by the Database and Artificial Intelligence Group of the Vienna University of Technology. The project has partly been funded by the Austrian Science Fund (FWF)[2] under grant P12344-INF. The main goal of the JADE project is to extend and examine the application of model-based diagnosis techniques to programming languages in general. In particular, object-oriented languages are of interest due to their widespread usage and their position at the forefront of programming language research. This includes theoretical research about software modeling and the debugging of computer programs and practical issues, such as the implementation of various models and an interactive debugging tool. Whereas all implementations are done in Smalltalk, the target programming language was chosen to be Java.

4.1 The Java programming language

Java (see [16, 20, 47]) is a general-purpose object-oriented programming language, which is based on a variety of other programming languages, such as C, C++, Objective-C, Smalltalk, Lisp, and Modula-3. It was originally developed by James Gosling at Sun Microsystems under the name *Oak* and designed for the development of applications for consumer electronics, such as TV-top boxes. In 1995 the language was renamed Java, when Gosling and his colleagues were moving away from the hardware aspect of the language to a more powerful general-purpose programming language designed for efficient networking and Internet communications and the creation of powerful GUIs. With the rise of the Internet Java became one of the most prominent programming languages. One of its main advantages is its high portability, which is a result of the fact that Java programs are compiled into a platform-independent byte-code. This byte-code can then be executed by a Java Virtual Machine (JVM). Due to the exact specification the JVM [23] Java systems run on nearly all existing platforms and

[1] http://www.dbai.tuwien.ac.at/proj/Jade/
[2] http://www.fwf.ac.at

hardware environments, which explains the language's success, especially in the area of Internet applications.

As already mentioned, the Java programming language [16] is used in the JADE project as the target programming language. This means that in a first step models of valid Java programs have to be created, which cover all source code structures required for an efficient debugging process. In a second step the Java source code is then debugged using the created models. There are multiple reasons why Java has been selected as the target programming language of the JADE project. Among these are:

- Java is a general-purpose programming language with widespread usage in various software engineering areas. Moreover, its usage is still predicted to be increasing in the coming years.
- Java is strongly typed and has relatively simple semantics, in particular compared to C++, the most widespread object-oriented programming language. It therefore seems to be better suited to the application of MBD techniques to the debugging of object-oriented languages in this relatively new research area.
- Java provides specific features for developing and running applications over the World Wide Web (WWW), which is expected to present new challenges for the MBD approach.
- Java is also expected to be widely used in the future by people with little programming experience as it becomes the language of choice for developing small, ad-hoc WWW applications. In this context, a knowledge-based debugger that guides an inexperienced user through the debugging cycle would be especially important.

4.2 Goals of the JADE project

The goal of the JADE project has originally been stated as the examination of the formal underpinnings required for using MBD in a standard software development environment, and the development of an experimental system for solving concrete diagnosis problems in form of Java programs. In particular, the JADE objectives can be summarized as follows:

1. Development of a theory of model-based software debugging.
2. Description of Java semantics in terms of logical models usable for diagnosis.
3. Examination of the use of alternative models for scalable diagnosis and their flexible use in a multi-model diagnosis environment.
4. Development of an intelligent debugging environment for Java programs based on the theoretic results.

5. Collection of example programs and establishment of a set of benchmarks for diagnosis evaluation.

4.3 Current state of the JADE project

The JADE project is divided into several parts, each considering some aspects of the problem. In the following we describe the current state of the JADE project for each individual activity:

Theoretical aspects: This part of the project aims at the development of a theory of MBSD, which includes the examination of different approaches to the modeling of programming language constructs, in particular with regard to object-oriented languages like Java. Current results of this activity have been entered into the design of the diagnosis system for debugging Java programs during all stages of the project. A variety of publications can also be seen as the result of this part of the project (see [44, 40, 41, 28, 29, 31, 30, 25, 24, 26, 27]).

Applying MBSD principles to Java programs: The use of MBSD for Java requires the description of Java semantics in terms of logical models usable for diagnosis. So far, two model families have been developed, i.e., functional dependency models and value-based models. Both model families have been implemented and tested in various forms. This work focuses on the creation and application of various functional dependency models. Discussions about value-based models as well as first debugging results with these models can be found in [31] and [32]. Currently, both model families cover a large subset of the Java programming language. The extension to yet uncovered language features, e.g., exception handling, is still open to further research.

Development of an intelligent debugging assistant: This is the most strongly implementation-oriented part of the project. It deals with the development of an intelligent debugging environment for Java programs based on the theoretical results, again, obtained from the JADE project. Currently, a debugging tool is in use, which deals with both model families and allows for an interactive fault localization process. The JADE debugger makes use of a code instrumentation module, which computes evaluation traces of Java programs by instrumenting the source code and running the program on a standard JVM. Work on this part of the project proceeds in parallel with the more theoretical stages to guarantee that the environment is ready for use when needed for experimenting with different diagnosis models and algorithms. The JADE debugger is shortly described in Chapter 19.

Accumulating a set of example programs: Evaluation and testing of the diagnosis algorithms requires the compilation of a set of example programs of different size, complexity, and application domain. This set of programs

is used as the base of a set of benchmarks for the (performance- and quality-related) evaluation of different diagnosis approaches. Currently, there exists a JADE test-case suite, which includes both, small demonstration programs, which demonstrate certain language constructs and modeling features, and medium-size applications[3]. The used models have been tested on the complete JADE test suite. Empirical results obtained from tests with all sorts of functional dependency models can be found in Chapter 20. The creation of larger, real-world applications and tests with new test programs are left for future research.

The software models described in the following chapters (see Part II) and all results of concrete debugging sessions (see Part III) directly build on the theoretical basis provided by the JADE project. All models described herein have been implemented in the course of the JADE project and been tested on valid Java programs. Note that in the following chapters some theoretical descriptions are supplemented with references to problems and solutions of the JADE project. These sections are especially labeled with the keyword JADE. All empirical results stated in Part III of this work were obtained from experiments with the JADE debugging environment, which is briefly described in Chapter 19.

[3]The current version of the JADE test suite can be downloaded from the JADE project page (http://www.dbai.tuwien.ac.at/proj/Jade/)

Part II

Modeling

Chapter 5

Modeling Java Systems

Since every model-based approach requires a model of the analyzed system, Part II of this work is dedicated to the definition and creation of different models of `Java` programs, which are suitable for debugging. We start off with a short analysis of software models and `Java` systems in general.

5.1 Software modeling

Generally speaking, a model is an abstract description of a given real-world system. According to [51] a model can be a physical, mathematical, or logical representation of a system, entity, phenomenon, or process. More formally, a model is any system specification, which in most cases consists of a set of instructions, rules, equations, or constraints for generating I/O behavior. [51] defines a general framework for modeling, which consists of the following elements:

The source system is the real-world or virtual environment that we are interested in modeling. It can be seen as a *source of observable data*, from which the *system behavior database* is deducted. The latter contains all data gathered from observing or experimenting with the system.

The experimental frame is a specification of the conditions under which the system is observed. As such, an experimental frame is the operational formulation of the objectives that motivate a modeling project.

The model itself is a system specification, which through a set of instructions, rules, equations, or constraints, exhibits a well-defined I/O behavior.

The simulator is a system capable of executing a model in order to generate its behavior.

In this work we are mainly interested in the modeling of software systems or, to be more precise, the modeling of `Java` programs. The objective of creating such a model is to efficiently locate faults in the `Java` system. In this context the above framework can be stated a bit more precisely:

The source system we are observing is a concrete `Java` system. This includes not only the source code, but also a `Java` compiler, the created byte-code, the `Java` Virtual Machine (JVM), and finally the run-time behavior of the system. Note that other components like the hardware and the operating system also play a crucial role in the observation of a given `Java` system. The content of the *system behavior database* depends on the exact type of model and the objective of modeling. Since we are ultimately interested in source code fault localization, we capture all components, which might lead to a system failure.

The experimental framework is, again, determined by the goal of source code fault localization. As we are not interested in the detection of hardware faults or compiler bugs, our experimental frame purely focuses on the source code. It is assumed that all other system components, i.e., hardware, operating system, compiler, JVM, etc..., exhibit an expected behavior. We further assume that the `Java` source code passes all syntactical checks by the compiler and the system terminates on the given source code in all cases and does not produce any run-time errors.

The model is an abstract representation of the analyzed `Java` system. Of course, there exists a wide variety of program models, which can be used for debugging. In the following sections some models are described, which are based on the collection of functional dependencies.

The simulator is a system, which uses the produced model for debugging. What such a system looks like and how good it is in finding source code faults is described in Part III of this work.

Of course, there exist many different types of software models. Roughly they can be divided into multiple categories using the following criteria:

Granularity: Models can be created for methods, individual blocks, statements, or expressions. The models described herein are mostly method models, which consist of models of all their statements. Expression models are in most cases not explicitly created, which leads to a debugging process at statement and not at expression level.

Used information: Models can further be divided by the amount of information used during their creation. Purely static models only make use of information, which is known at compile-time, i.e., the source code and well-defined programming language semantics. Dynamic models, i.e., value-based models, explicitly simulate run-time behavior by propagating concrete values through the system and thus evaluating the program. A third approach is to use static information together with an evaluation trace. By doing this we can incorporate dynamic information into a static model and thus make the resulting model more powerful. Examples of all kinds of models are presented in the following sections.

Hierarchy: Models can be constructed as hierarchical models, i.e., a model component contains sub-models, which specify certain parts of the source system in more detail. Models, which do not make use of hierarchical model components, have to include all necessary information in their top-level structure.

Models of object-oriented programming languages like `Java` are in most cases more complex than models of functional or purely imperative languages (see [42, 43, 50]). This is, because of the inherently more complex system structure of object-oriented systems. Among others models of these systems have to deal with the following problems:

- Imperative features, e.g., selection statements, loops, method calls, strings, arrays, etc...
- Object-oriented features, e.g., multiple classes and objects, inheritance, polymorphism, etc...
- Method calls with side-effects, which is very common in object-oriented programming
- Efficient handling of aliasing problems
- Recursive method calls (direct and indirect recursion)

Finally, we state some requirements as they apply to all kinds of models. In the following sections we then show, which of our models meet these requirements:

Soundness: A model is said to be *sound*, if the model correctly represents all aspects of the modeled source system. If, for example, we want to create a model including all data dependencies between different variables in a given method m, a sound model never produces a dependency, which cannot be found in m. In the following chapters the goal is to produce sound models. However, we will see that it cannot be guaranteed in the general case that only sound models are created.

Completeness: A model is said to be *complete*, if the model covers all aspects of the source system lying in the model's scope. In the above example this means that we expect the model to create all dependencies between variables of m and not just a few ones. As incomplete models cannot guarantee to work in all aspects, completeness of the constructed models is another goal in the next couple of chapters.

Minimality: A model is said to be *minimal* if it does not contain more elements than needed for fulfilling its objectives. A model creating dependencies, which are generally correct, but not needed for a particular case, is not minimal in respect to this case. We come back to the concept of minimality in the following chapters.

5.2 Java systems

Before discussing the creation of models in detail we have a short look at the source system of our modeling process: the Java system. A Java system consists of a wide variety of system components, such as the used hardware, compilers, source code files, system settings, operating system, the Java Virtual Machine (JVM), etc... As we are only interested in locating bugs in source code files, we merely look at those parts of a Java system, which are directly relevant to our goal. In particular, these are:

- All source code files, i.e., compilation units of a given Java system.
- The Java programming language semantics as defined in [16].
- The input/output behavior of the system at run-time. This includes all test-cases mapping an input vector to both, a computed and an expected output vector.
- Additional run-time information like the values of certain variables during program execution. As mentioned above some models make use of the evaluation trace taken directly from the Java system.

In order to keep the resulting models as simple and small as possible, we abstract from all system features, which do not lie within the scope of our debugging goal. Therefore, we formulate the following assumptions, which are valid during the whole modeling and debugging process:

- All used hardware and low-level software components, e.g., operating system, network, drivers, etc..., work correctly
- The same is true for the used Java compiler and JVM.
- All source code compilation units are syntactically correct, i.e., they pass the Java compiler without error messages.
- The compiled byte-code can be executed by the JVM and terminates on all inputs. The debugging of faults leading to infinite loops or method calls is not within the scope of this work.
- No run-time failures occur during program execution. At the moment this also includes exceptions, which as we will see cannot yet be handled by the proposed models (see Section 10.6).

To demonstrate the properties of a Java system and their effects on the resulting models we implement a short example program *Point.java*, which provides the basic data structure and functionality of a two-dimensional point (see Figure 5.1). We use this example throughout this chapter as a running example.

```
      class Point {
        int x;
        int y;

        Point(int x, int y) {
1.        this.x = x;
2.        this.y = y; }

        Point plus(Point p) {
1.        return new Point(x+p.x, y+p.y); }

        public static void test() {
          Point p1, p2;
1.        p1 = new Point(0,0);
2.        p2 = new Point(2,3);
3.        p1.x = 1;
4.        p1.y = 2;
5.        p2 = p1.plus(p2); }
      }
```

Figure 5.1: Example program *Point.java*

5.3 Compile-time description of Java systems

At compile-time a syntactically correct Java system consists of multiple files, i.e., compilation units, each of which stores the source code of various Java classes. The resulting Java host environment can logically be seen as a set of packages, which include classes, interfaces, and sub-packages. In particular, a host environment consists of the following hierarchical levels:

The Host environment is the top-level of the Java system containing a set of packages.

Packages include classes and interfaces of the Java system. Further on, a package may include sub-packages, what allows for a hierarchical package structure in Java.

Classes and interfaces are the main data structures in any object-oriented programming language. Each class consists of multiple instance and class field definitions, static initializers, and method declarations (including constructors). Note that since version 2 of the JLS [16] inner classes are also part of the Java programming language.

Methods consist of hierarchically nested blocks. The method body can be seen as the top-level of the nested block structure.

Blocks are ordered collections of statements.

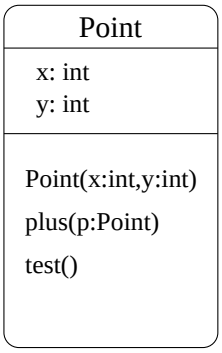


Figure 5.2: UML representation of class *Point*

Statements include expressions and sub-blocks

Expressions contain constants, i.e., literals, variables, and sub-expressions

There exist multiple techniques and standards of describing the static appearance of `Java` host environments. Currently, the most important standard is the Unified Modeling Language (UML) (see [36, 34]), which specifies the static properties of a given system through the use of various diagrams, e.g., class diagrams, object diagrams, sequence diagrams, collaboration diagrams, state charts, etc... A UML representation of class *Point* is given in Figure 5.2.

In the following chapters we are mainly interested in the modeling of `Java` systems at method level down to the modeling at expression level. We therefore give the following definitions:

Definition 5.3.1 *$CLASSES$ is defined as the set of all classes of the `Java` host environment under consideration.*

Definition 5.3.2 *$METHODS$ is defined as the set of all method and constructor declarations of the `Java` host environment.*

Definition 5.3.3 *$BLOCKS$ is defined as the set of all (nested) blocks of the `Java` system.*

Definition 5.3.4 *$STATEMENTS$ is defined as the set of all statements of the `Java` system.*

Definition 5.3.5 *$EXPRESSIONS$ is defined as the set of all expressions of the `Java` system.*

Definition 5.3.6 *$CONSTANTS$ is defined as the set of all constants of a given `Java` host environment. By constants we mean literals, which are defined in the source code and never change their value during execution. Logically, constants can be seen as objects of pre-defined type, which do not change their internal state.*

Definition 5.3.7 *VARS is defined as the set of all variables (of primitive or of reference type) defined in the Java system.*

Definition 5.3.8 *LOCATIONS is defined as the set of all class instances, which might be created at run-time. At compile-time, we can determine all locations by looking at all class instance creation expressions of a given Java system. Note that class instances may also arise from string literals, which at run-time are automatically promoted to instances of type string.*

Note that all program structures as defined above are used in this work relative to each other, e.g., $BLOCKS_m$ with m being a method denotes all blocks defined in the body of m rather than in the whole host environment.

One of the key components of all models proposed in the following chapters are Java variables. They can be classified by their:

Types: $v \in VARS$ can be of primitive or of reference type. Primitive variables, e.g., x and y in Figure 5.1, hold values, reference variables, e.g., *p1* and *p2*, reference objects, i.e., they point at a particular memory location.

Position in the source code: We distinguish (1) class variables, which are defined once for a particular class of the system, (2) instance fields, which exist for each instance of a given class, and (3) local variables, which are only visible inside the blocks (and all sub-blocks) where they are defined.

Example 5.3.1 *In the example depicted in Figure 5.1 we find only one class, i.e., class Point with two instance fields, one constructor, and two method declarations. Class Point could be described by* $METHODS = \{Point(int\ x, int\ y), plus(Point\ p), test()\}$*, where, for instance,* $BLOCKS_{test()} = \{b\}$ *with* $b = \{s_1, s_2, s_3, s_4, s_5\}$*. Method test() defines two local variables of reference type, p1 and p2, and makes use of the instance fields x and y of instances of class Point.*

5.4 Run-time descriptions of Java systems

When we try to describe a certain Java system at run-time, we find that the description looks somehow different from the one in Section 5.3. First of all we distinguish between descriptions of a Java system from a static and from a dynamic point of view. Descriptions from a dynamic point of view make use of a concrete evaluation of one of the system's methods and therefore cover the non-ambiguous state of the system at run-time. Descriptions from a static point of view are a bit harder to specify. Whereas only information about the system's source code and a priori language semantics is available (hence the term static point of view or static description), the concrete state of a Java system during run-time has to be covered. The following sections introduce run-time descriptions from both, static and dynamic viewpoints.

5.4.1 The dynamic viewpoint

From a dynamic point of view it is quite easy to define a snapshot of a particular Java system, because we can use the information provided by an evaluation trace. This evaluation trace contains elements, which are not known at compile-time and therefore cannot be part of a purely static system description as in Section 5.3. We define:

Definition 5.4.1 *$VALUES \supseteq CONSTANTS$ is defined as the set of all values, which are produced by the Java run-time system in all possible evaluation traces. This definition covers literals and other values of primitive type that are computed at run-time.*

Definition 5.4.2 *$OBJECTS$ is defined as the set of all object locations created by the Java run-time system during program execution.*

The following components have to be defined in order to describe the state of a Java system at an arbitrary point during program execution:

Values: The set $VAL \subseteq VALUES$ of all values, which are assigned to Java variables during the execution of a method of the Java system. These values can either be part of the source code in form of literals (in our example the integer values 0, 1, 2, and 3) or be created by the Java run-time system (the integer value 5 in our example). Therefore, VAL may include values of variables of primitive type, i.e., booleans (`true`, `false`), numeric values (**byte**, **short**, **int**, **long**, **char**), and floating point values (**float**, **double**).

Objects: The set $O \subseteq OBJECTS$ of object locations which store the state of a particular Java object. Objects and their memory locations are produced in the course of program execution either by explicit constructor calls or implicit object creations performed by the Java run-time system, e.g., string promotions. The internal state of object locations is user-defined and alterable. In our example 3 objects of type *Point* are created.

Variables: The set VV where $vv_i \in VV$ is a tuple $\langle v, d \rangle$. $v \in VARS$ is a variable and $d \in VAL \cup O$ is a concrete value assigned to or an object referenced by variable v.

At run-time the exact state of a Java system at a particular point in execution, i.e., after the execution of statement n of method m, can now be defined by the tuple $system^n_m = \langle O, VV \rangle$. A snapshot of our example program after the execution of all five statements of method *test()* reads as follows:

$$system^5_{test()} = \langle \{o_1, o_2, o_3\}, \{\langle p1, o_1\rangle, \langle p2, o_3\rangle, \langle o_1.x, 1\rangle, \langle o_1.y, 2\rangle, \langle o_2.x, 2\rangle, \langle o_2.y, 3\rangle, \langle o_3.x, 3\rangle, \langle o_3.y, 5\rangle\}\rangle$$

where o_i stands for an instance of class *Point*. Figure 5.3 shows the state $system^5_{test()}$.

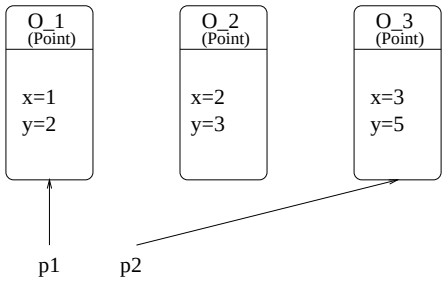


Figure 5.3: The Java system $system^5_{test()}$

5.4.2 The static viewpoint

Obviously, we cannot define the run-time state of a Java system at an arbitrary position in its evaluation by only looking at the source code. This is, because values and objects are created at run-time and can only be predicted to a certain extent at compile-time. However, there is still a lot of information about a system's run-time behavior, which can be approximated at compile-time. In the following we give a static description of a Java system, which can serve as an approximation of the dynamic description proposed in Section 5.4.1.

Constants: The set $C \subseteq CONSTANTS$ of literals, that are defined in the Java source code and do not change their values during program execution. Although in Java these structures are no objects, they can logically be seen as objects with a fixed content of primitive type (in our example we find the constants 0, 1, 2, and 3). Note that each occurrence of a constant has to be considered separately. In this work we omit indices of constants for simplicity.

Locations: The set $L \subseteq LOCATIONS$ of memory locations, which store the state of a particular Java object. Locations are produced by constructors or by default. Their internal state is user defined and alterable (in our example three objects of type *Point* are created). Note that in contrast to the dynamic viewpoint, the exact run-time type of a certain location cannot always be determined by a purely static analysis. Therefore, locations represent a more abstract view of concrete class instances than objects used in a dynamic system description. Section 6.4 describes the different types of locations, which are used during the creation of all models, in more detail. Some model types make use of *multiple-locations*, which represent multiple concrete locations by a single model component. These locations provide an even more abstract view of class instances created at run-time.

Variables: The set VV' of tuples mapping variables to a set of possible values, i.e., $vv'_i \in VV'$ is of the form $\langle v, D \rangle$. $v \in VARS$ is a variable and $D \subseteq CONSTANTS \cup LOCATIONS \cup VARS$ is a set of constants, variables and abstract representations of class instances possibly influencing the current value of variable v or the state of the object referenced by v.

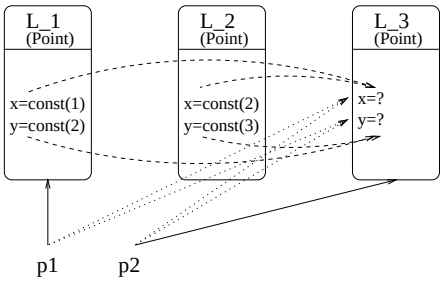


Figure 5.4: Approximation of the Java system $system^5_{test()}$

The state of a Java system at any point of execution can now be approximated by $system^n_m = \langle L, VV' \rangle$. A snapshot of our example program after execution of method *test()* reads as follows:

$$\begin{aligned} system^5_{test()} = \langle \{l_1, l_2, l_3\}, \{ & \langle p1, \{l_1\}\rangle, \langle p2, \{p1, l_3\}\rangle, \langle l_1.x, \{p1, 1\}\rangle, \\ & \langle l_1.y, \{p1, 2\}\rangle, \langle l_2.x, \{2\}\rangle, \langle l_2.y, \{3\}\rangle, \\ & \langle l_3.x, \{p1, p2, l_1.x, l_2.x\}\rangle, \langle l_3.y, \{p1, p2, l_1.y, l_2.y\}\rangle\}\rangle \end{aligned}$$

Figure 5.4 shows the approximation of the state of our example program after the execution of all five statements of method *test()*. We can see the current state of the three objects (memory locations) given by their instance fields which are either assigned to constants or run-time values. Whereas all constants are known at compile-time, the run-time values can only be approximated by the set of all language constructs, which are known to have an influence on the current value of the instance fields. The two variables of reference type, i.e., *p1* and *p2*, reference the first and third location, respectively. The object at location 2 is not referenced by any of the system's variables and is eliminated by the garbage collector.

Chapter 6

Functional Dependency Models

The notion of functional dependencies (FDs) within computer programs and the analysis of functional dependency graphs (FDGs) have been well-known concepts in the software engineering community for many years. Informally, a FD describes the fact that the value of a certain variable at a particular point within the program depends on some other variable values, constants, or methods. In the following we will discuss the concept of FDs in more detail.

6.1 Variable occurrences

First of all it is important to note that each variable v defined in a specific method m can change its value several times during the execution of m. This happens as a consequence of an assignment statement or a method call, which may alter the value of v through side-effects. In order to non-ambiguously identify a certain variable at a particular point within a given block of statements b, we assign indices to all occurrences of variables in b starting with index 0. If a variable v appears on the left-hand side of an assignment, its index is incremented by one. These indices are unique relative to the block b, in which v appears. We no longer speak about variables, but about variable occurrences. More formally, we define:

Definition 6.1.1 *A variable occurrence VO is a tuple $\langle v, b, i \rangle$ where*

- *$v \in VARS$ is a variable defined in the `Java` system*
- *$b \in BLOCKS$ is a block of the `Java` system*
- *$i \in N_0$ is a unique index relative to block b*

Example 6.1.1 *If we look at statement line 1 of method test() in Figure 5.1, we find a variable occurrence $\langle p1, b, 1 \rangle$, which refers to the point in the top-level block b of the body of method test(), where the value of variable p1 is changed for the first time. Here, the variable occurrence appears on the left-hand side of an assignment statement.*

```
      Point maxCoordinate() {
        int tmp;
1.      if (x > y)
1.1.      tmp = x;
        else
1.2.      tmp = y;
2.      return tmp;
      }
```

Figure 6.1: Example method *maxCoordinate()*

Note that all VOs should be unique. Following the above definition this is the case if all blocks of the system are given unique names. It is worth mentioning that most `Java` methods consist of hierarchically nested blocks. VOs have thus to be defined hierarchically for each block as well. For instance, look at method *maxCoordinate()* of class *Point*, which is depicted in Figure 6.1. It returns the maximum of the two coordinates of a given point object.

Here, we get two VOs of variable *tmp* ($\langle tmp, t, 1\rangle$ and $\langle tmp, e, 1\rangle$ for the two branches t and e of the selection statement in line 1. At the top-level of statement line 1 we get only one VO, i.e., $\langle tmp, m, 1\rangle$, where m is the top-level block of method *maxCoordinate()*.

Variables can also change there values due to side-effects of called methods. Therefore, VOs can also arise from method calls, which can be demonstrated using statement line 1 of method *test()* (see Figure 5.1). Apart from the VO at the assignment's left-hand side, we get two more VOs arising through side-effects of the method call on the right-hand side, i.e., the constructor invocation of class *Point*. This is, because the two instance fields of the new point object (location 1) are set to initial values. The resulting VOs are $\langle 1.x, m, 1\rangle$ and $\langle 1.y, m, 1\rangle$, where m denotes the block containing the method call. As we will see, VOs are key components of all functional dependency models. Therefore, we give the following two definitions:

Definition 6.1.2 (Equivalence (VO)) *Let* $vo_1 = \langle v_1, b_1, i_1\rangle$ *and* $vo_2 = \langle v_2, b_2, i_2\rangle$ *be two VOs.* vo_1 *and* vo_2 *are said to be equivalent, i.e.,* $vo_1 = vo_2$*, iff* $v_1 = v_2 \wedge b_1 = b_2 \wedge i_1 = i_2$ *holds.*

Definition 6.1.3 VO_b *is defined as the set of all variable occurrences of block* b*.*

Furthermore, we can distinguish two different types of VOs depending on their syntactic positions within the source code. A variable appearing on the left-hand side of an assignment is said to be set to a certain value. Only assignments can change the value of a variable at a particular VO. In all other cases variables are simply used, i.e., their value is used in the course of the evaluation of expressions and statements. The following definitions should clarify this distinction:

Definition 6.1.4 *A variable occurrence* $vo \in VO_b$ *is called assignment variable occurrence (AVO) if it appears on the left-hand side of an assignment. This*

includes all assignments in called methods, which are imported into the calling method through side-effects. The set of all AVOs of block b is called AVO_b.

Definition 6.1.5 *A variable occurrence* $vo \in VO_b$ *is called used variable occurrence (UVO) if it does not appear on the left-hand side of an assignment. The set of all UVOs of block b is called* UVO_b.

Obviously, the following proposition holds:

Proposition 6.1.1 $AVO \cup UVO = VO$

Example 6.1.2 *If we come back to method maxCoordinates(), we find three UVOs at the method's top-level block m (*$\langle x, m, 0\rangle$ *and* $\langle y, m, 0\rangle$ *in line 1, and* $\langle tmp, m, 1\rangle$ *in line 2), but only one AVO (*$\langle tmp, m, 1\rangle$ *in line 1). Clearly, only the value of variable tmp is changed, whereas all other variables are unaltered.*

6.2 Functional dependencies

We can now compute one FD for every AVO in our system, i.e., whenever a variable changes its value exactly one FD arises from the associated AVO. Let $x \in AVO_b$ and $y \in VO_b$. We say that x depends on y iff a change of the value of y results in a change of the value of x. In other words, x depends on y iff there is an execution $e1$ of b such that if at point t during $e1$ we alter the value of the variable occurrence y (thus producing a new execution $e2$), then there is a later point t' such that the value of x in $e1$ is different from its value in $e2$.

We can collect the set V of all VOs, on which a given VO, e.g., $vo \in AVO$, depends. Often we are not only interested in VOs influencing a vo, but also in other system components, whose character determines the current value of vo (e.g., constants, called methods, etc...).

Definition 6.2.1 *Generally, DEP is defined as the set of all system components having an influence on the value of a given AVO, i.e.,* $vo = \langle v, b, i\rangle$.

So far, DEP only contains VOs, i.e., $DEP \subseteq VO_b$. We write $DEP = \langle V\rangle$, where V is the set of all VOs influencing the value of vo. Generally, DEP not only contains VOs, but also other system components, such as constants, run-time values, objects, method declarations, etc... This is, why we give a general definition of DEP, which will be extended in the following chapters, when actual model types are introduced. In this section, for the sake of clarity, we assume that only VOs have an influence on a given VO, i.e., DEP only contains VOs. Using the definition of DEP, we can now formally define a functional dependency (FD) of a given VO, e.g., $vo \in AVO$.

Definition 6.2.2 *The tuple* $\langle vo, DEP\rangle$ *is called a functional dependency of the variable occurrence* $vo = \langle v, b, i\rangle$, *i.e.,* FD_{vo}.

Example 6.2.1 *Consider the following source code fragment, which sets the value of variable x:*

1. *x = 3*a + y.m();*

We can now compute a single FD for the AVO $vo = \langle x, b, 1\rangle$ *with b being the block containing statement line 1. If we only consider VOs as constituents of DEP, we get* $DEP =< \{\langle a, b, 0\rangle, \langle y, b, 0\rangle\} >$ *and* $FD = \langle vo, DEP\rangle$*. As already mentioned other components (e.g., constant 3, method m(), etc...) could also be taken into consideration during the computation of DEP.*

Intuitively, it seems clear that DEP contains all program components, which influence the value of a particular VO, e.g., vo. Ideally the elements of DEP model all program structures, which influence vo at run-time and thus include not only static features, as variables and method declarations, but also dynamic structures, such as values computed by the JVM at run-time and objects created in the course of program execution. The problem with a purely static analysis is that these run-time values and objects cannot always be predicted at compile-time. Therefore, the exact look and interpretation of the various parts of a FD also depends on the exact model created and even more on the run-time information used during the modeling process. The above definition serves as a general framework for all FDs used in this work. In the following sections about concrete types of FDMs the meaning of individual parts of a FD is specified more precisely. Finally, we give two definitions of the equivalence of FDs and DEPs. This is done at this stage, because it is needed later on, e.g., for the modeling of recursive methods (see Chapter 13).

Definition 6.2.3 (Equivalence (DEP)) *Let* $DEP_1 = \langle V_1\rangle$ *and* $DEP_2 = \langle V_2\rangle$ *be two DEP structures.* $DEP_1 = DEP_2$ *holds iff* $V_1 = V_2$*.*

Definition 6.2.4 (Equivalence (FD)) *Two FDs,* FD_1 *and* FD_2*, are said to be equivalent, iff both, their left-hand side and their-right hand side, are equal. More formally, let* $FD_1 = \langle vo_1, DEP_1\rangle$ *and* $FD_2 = \langle vo_2, DEP_2\rangle$ *be two FDs. Then* $FD_1 = FD_2$ *iff* $vo_1 = vo_2$ *and* $DEP_1 = DEP_2$*.*

6.3 Types of dependencies

Until now we were talking about functional dependencies, but did not consider the various kinds of dependencies. In [21] a classification of dependencies between program instructions is given, which distinguishes between (1) data influences, (2) control influences, and (3) potential influences. In this section we adapt this classification to dependencies between variable occurrences. The resulting classification, which will be used throughout this work can shortly be described as follows:

Data dependencies or *data influences* between variable occurrences vo_1 and vo_2 occur, if vo_1 is the left-hand side of a variable assignment and vo_2 appears in the assignment's right-hand side. The value of vo_1 is said to depend on the value of vo_2.

Control dependencies arise in loop or selection statements, where the evaluation value of a condition determines the instructions to be executed at run-time. A control dependency or *control influence* between variable occurrence vo_1 and vo_2 occurs, if the value of vo_1 is changed in the branch of a selection statement or the body of a loop statement and vo_2 appears in the loop or selection statement's condition. As we will see control dependencies are created by combining multiple sub-models (branch models, body models) into a single statement model, whereas data dependencies arise directly from variable assignments. With the exception of conditional expressions of the form $expr_1 \,?\, expr_2 \,:\, expr_3$ control influences only appear at statement level.

Potential dependencies or *potential influences* are purely static concepts, which ignore run-time dependencies and focus on all variable influences, which possibly occur at run-time. Imagine a data dependency between the variable occurrences vo_1 and vo_2 arising from a variable assignment in one branch of a selection statement. If at run-time this branch is not executed, the abovementioned data influence never arises. Nevertheless, from a static point of view it constitutes a potential dependency, which might effect the value of vo_1 at run-time. Note that in the following two sections two concrete FDMs will be introduced. Whereas the first one (*ETFDM*) makes use of run-time information and can therefore determine all data and control dependencies arising at run-time (see Chapter 7), the second model (*DFDM*) is a purely static model, which (due to a lack of run-time information) takes all possible run-time scenarios into account. The latter model therefore includes potential influences (see Chapter 8).

6.4 Locations

In order to model objects influencing a certain variable v we use locations, which can be seen as an abstract representation of a class instance residing at a particular memory location. In this section we introduce the basic concepts and types of locations, which are used in all proposed models either implicitly or explicitly. The exact interpretation of locations in different models and their usage during the modeling of complex program structures are explained in following chapters.

Each location l has a type, i.e., the class type of the instance modeled by l, and a unique identifier. In our case we chose a unique index, which non-ambiguously identifies location l within a modeled method m. More formally, we write:

Definition 6.4.1 *A location l is defined as a tuple $\langle type, index \rangle$, where $type \in CLASSES$ and $index \in N_0$.*

We distinguish the following types of locations:

New locations arise from class instance creation expressions. In case of new locations the exact run-time type is known, which is always equivalent to the type of the class instance creation expression. In method *test()*

(see Figure 5.1) two new locations arise from the constructor invocation expressions in lines 1 and 2, respectively.

Default locations are created during the modeling of method m, if an object that has not been explicitly created by the call of a class instance creation expression, is accessed through a reference variable v, e.g., a field access or a parameter of m. This can be the case, if v which is visible outside of m is assigned a class instance in another method or static initializer. In case of default locations the exact run-time type cannot always be determined. Its type is initially set to the type of the reference variable v, but can also contain any sub-type due to inheritance. Note that in method *test()* no default locations appear.

Imported locations: If a new or default location l is created in method n, but imported into method m through a method call of n in m, l will become an imported new location or an imported default location, respectively. A more detailed description of imported locations is given in the context of modeling method calls (see Section 9.2). In method *test()* one new location is created in the called method *plus(Point p)* and imported into method *test()*.

Multiple-locations: All locations described above can be marked *single* (by default) or *multiple*. If a location l is marked as multiple, it no longer represents only one class instance but a set of n objects of the same type. Note that the number of instances modeled by l may be unknown. The use of multiple-locations is described in Section 10.4 and Chapters 11 and 13.

In order to uniquely identify a location l we assign it a location key. The location key for the individual types of locations is defined as follows:

New locations keys: A new location l can uniquely be identified by the class instance creation expression for which is was created. We define $key(l) = \langle pos \rangle$, where *pos* is the source code position of the class instance creation expression.

Default locations keys: As already mentioned a default location l is created for a reference variable v accessing the object modeled by l. We therefore define $key(l) = \langle v \rangle$, where $v \in VARS$ is a variable of reference type initially pointing at l.

Imported location keys: The keys of imported locations consist of two parts: (1) the key of the original new or default location and (2) a method call path storing information about called methods and method calls, through which l has been imported into the currently modeled method. An exact definition of imported locations and their keys is given in Section 9.2.

Multiple-location keys: The keys of multiple-locations do not differ from normal new location or default location keys.

In the next couple of sections we introduce various types of FDMs. All these models either contain locations or use locations during their creation. As we will see the concept of locations is the key to handle aliasing problems. Note that in all example FDMs we denote locations by their unique numbers and, for the sake of clarity, do not state the exact type of the location.

6.5 Functional dependency models

So far we have defined FDs for single variable occurrences. These FDs arise from either assignments or method calls, both of which are expressions following the `Java` language specification. We therefore speak of assignment and method call FDs, respectively. In the following sections we show a complete functional dependency model (FDM) for expressions, statements, methods, and even for whole host environments.

6.5.1 FD expression models

In order to compute a full FD model of a given expression (possibly including cascading assignments and multiple method calls), we have to compute the FDs arising from all AVOs of that expression, i.e., determine all assignment and method call FDs. Formally, we get

Definition 6.5.1 *FDM_e, i.e., the FDM of expression $e \in EXPRESSIONS$, is defined as the set $\{FD_{vo} \mid vo \in AVO_e\}$*

Example 6.5.1 *Consider statement line 5 of method test(). This statement contains three assignment variable occurrences, i.e., $\langle p2, m, 2\rangle$, $\langle 3.x, m, 1\rangle$, and $\langle 3.y, m, 1\rangle$. If we want to compute the FDM arising from the method call, i.e., $FDM_{p1.plus(p2)}$, we get two FDs representing the value changes of the two instance fields of the newly created location 3. We write*

FDM*p1.plus(p2)*

$St.5: \; 3.x_1 \leftarrow \{1.x_2, 2.x_1\}$

$St.5: \; 3.y_1 \leftarrow \{1.y_2, 2.y_1\}$

Note that in case of conditional expressions of the form $expr_1 \; ? \; expr_2 \; : \; expr_3$ this is not always correct. The expression model in this case is computed similarly to the model of **if** statements (see Section 10.3). The creation of expression models will be handled in Chapter 9 in more detail.

6.5.2 FD statement models

If we want to create a FDM comprising all FDs of a certain statement s, there are the following three cases to be considered:

1. Statement s contains no expressions and no sub-blocks, e.g., empty statements, **break** and **continue** statements, variable declarations without assignments, etc... **Break** statements never give rise to any FDs.

2. Statement s contains an expression but no sub-blocks, e.g., **return** statements, unary pre- and post-operators. The model of these statements is the same as the FDM of the expression.

3. Statement s contains sub-blocks, e.g., nested blocks, **if** statements, loop statements, **switch** statements, etc... In this case the models of all sub-blocks are created first. The statement model is then computed by combining all sub-models. This procedure, however, is different for different statements and will be presented in more detail in Chapter 10.

Therefore, at statement level we no longer deal with assignment and side-effect FDs only, but also with selection and loop FDs, which are created through the combination of various FDMs of the statements' sub-blocks. Generally, a FD statement model reads as follows:

Definition 6.5.2 *FDM_s, i.e., the FDM of statement $s \in STATEMENTS$, is defined as the set $\{FD_{vo} \mid vo \in AVO_s\}$*

Example 6.5.2 *If we now want to compute a statement model for statement line 5. of method test(), i.e., $FDM_{p2=p1.plus(p2);}$, we collect all FDs arising from the assignment expression. As the statement contains no sub-blocks, the statement model equals the model of the assignment statement. The resulting model reads as follows:*

FDM$_{p2=p1.plus(p2);}$

$St.5: \; 3.x_1 \leftarrow \{1.x_2, 2.x_1\}$

$St.5: \; 3.y_1 \leftarrow \{1.y_2, 2.y_1\}$

$St.5: \; p2_2 \leftarrow \{p1_1, p2_1\}$

As mentioned above, the exact modeling of statements containing sub-blocks, especially selection and loop statements, is described in Chapter 10 in more detail.

6.5.3 FDMs of blocks and methods

Furthermore, we want to compute the FDM of a whole block of statements or even a whole method. The FD block model can be obtained by successively modeling all statements of the block in question. Formally, we write:

Definition 6.5.3 *FDM_b, i.e., the FD model of block $b \in BLOCKS$, is defined as the set $\{FD_{vo} \mid vo \in AVO_b\} = \{FDM_s \mid s \in b\}$*

Definition 6.5.4 *Let b be the top-level block of method m. Then the FDM of m is defined as the FDM of b, i.e., $FDM_m = FDM_b$.*

Example 6.5.3 *Computing the complete FDM of method test(), i.e., $FDM_{test()}$, we get the following model:*

$\mathbf{FDM}_{test()}$

$St.1: \ 1.x_1 \leftarrow \{\}$

$St.1: \ 1.y_1 \leftarrow \{\}$

$St.1: \ p1_1 \leftarrow \{\}$

$St.2: \ 2.x_1 \leftarrow \{\}$

$St.2: \ 2.y_1 \leftarrow \{\}$

$St.2: \ p2_1 \leftarrow \{\}$

$St.3: \ 1.x_2 \leftarrow \{p1_1\}$

$St.4: \ 1.y_2 \leftarrow \{p1_1\}$

$St.5: \ 3.x_1 \leftarrow \{p1_1, p2_1, 2.x_1, 1.x_2\}$

$St.5: \ 3.y_1 \leftarrow \{p1_1, p2_1, 2.y_1, 1.y_2\}$

$St.5: \ p2_2 \leftarrow \{p1_1\}$

6.5.4 FDMs of host environments

We define the FDM of a complete `Java` host environment as the sum of all method models of the `Java` system. Note that the computation of a host environment model only makes sense if we use a static model, which stays the same for all possible input vectors.

Definition 6.5.5 *The FDM of a given `Java` host environment is defined as the set $\{FDM_m \mid m \in METHODS\}$.*

6.6 Model views

6.6.1 Internal models

All models proposed so far are internal models, because they describe the internal dependency structure of a given expression, statement, block, or method. The following properties are unique to internal FDMs:

- FDs are computed not for variables, but for variable occurrences. In an internal model different occurrences of the same variable have to be distinguished.

```
1.  a = b;
2.  b = c;
3.  c = d;
4.  i++;
```

Figure 6.2: Example block b_1

```
1.  a = b;
2.  b = c;
3.  c = a;
4.  i++;
```

Figure 6.3: Example block b_2

- As a consequence, in internal models cyclic dependencies, i.e., FDs of the form x depends on y and y depends on x, cannot arise. This is, because we assign unique indices to all variable occurrences.
- There might exist multiple FDs for the same variable, though not for the same variable occurrence.
- Internal models contain FDs for local variables, which are only visible inside the modeled structure.

Internal FDMs can easily be visualized by the use of directed graphs (di-graphs). A digraph usually consists of a finite number of nodes, i.e., V, and a finite set of directed edges connecting two nodes, i.e., E. We denote a single edge connecting the nodes v_1 and v_2 by (v_1, v_2). If we now want to depict an internal FDM, we chose Algorithm 6.6.1, which works as follows:

Algorithm 6.6.1

- Create a node for each VO of the model, i.e., for each VO appearing on either the left-hand or the right-hand side of a FD in the FDM
- Create an edge (v_1, v_2) iff the FD $v_1 \leftarrow v_2$ is part of the FDM.

It follows from the above properties of internal models that the digraph of an internal model never contains cycles. Therefore, we speak about a directed acyclic graph (DAG).

Example 6.6.1 *Consider the two source code fragments given in Figures 6.2 and 6.3. The DAGs resulting from the blocks' FD method models are depicted in Figures 6.4 and 6.5.*

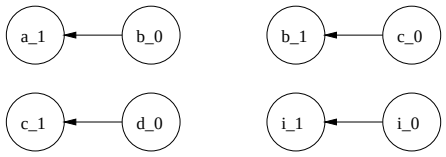


Figure 6.4: Internal model of block b_1

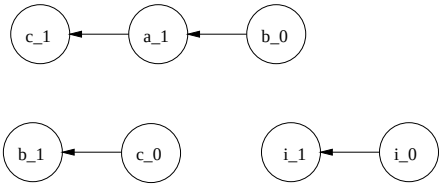


Figure 6.5: Internal model of block b_2

6.6.2 External models

Sometimes we are not interested in the internal view of a particular model, i.e., all FDs as they arise statement by statement, but only in the summarized model as it is visible from outside the modeled structure. We therefore shortly discuss external FD models for particular expressions, statements, or methods.

From an external view we are no longer interested in individual variable occurrences influencing each other, but only in dependencies arising from variables. Furthermore, we only consider FDs, which are visible from outside the modeled structure and discard all FDs of local variables. The following properties are unique to external FDMs:

- FDs are computed for variables, not for variable occurrences. In an external model different occurrences of the same variable are treated alike.
- As a consequence, in external models cyclic dependencies, i.e., FDs of the form x depends on y and y depends on x, can arise. This is, because multiple variable occurrences are contracted to a single variable.
- Only one FD for a given variable exists.
- External models do not contain FDs for local variables. Only variables, which are visible outside the modeled structure, i.e., input and output variables, are considered.

Therefore, external models contain a different type of FD, which herein will be called an external functional dependency (EFD). An EFD can be defined as follows:

Definition 6.6.1 *The tuple $\langle v, DEP \rangle$ is called an external functional dependency of variable v, i.e., EFD_v. Now DEP no longer contains variable occurrences influencing the value of v, but variables.*

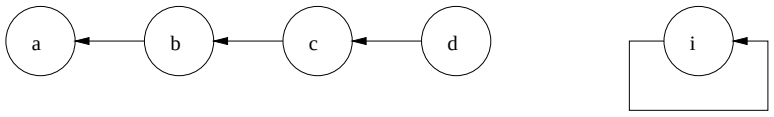


Figure 6.6: External model of block b_1

A single FD can be converted into an EFD by substituting all VOs by their variables. More formally, we define:

Definition 6.6.2 *Let $FD = \langle vo, DEP \rangle$, where $vo = \langle v, b, i \rangle$. Then the external view of FD, i.e., EFD, can be computed by $EFD = sum(FD) = \langle v, DEP' \rangle$, where DEP' equals DEP after all VOs have been substituted by their respective variables.*

When we now want to compute the summarized model of a complete FDM, i.e., $sum(FDM)$, this cannot be done by simply converting all FDs of FDM separately. Algorithm 6.6.2 defines, how $sum(FDM)$ is computed:

Algorithm 6.6.2 Let FDM be an internal functional dependency model of any source code structure (expression, statement, block, or method). Then the external FDM, i.e., $sum(FDM)$ can be computed as follows:

- Let EFDM be an initially empty external FDM
- Consider all FDs of FDM in the order in which they arise from the source code. Let the currently analyzed FD be $fd = \langle vo, DEP_1 \rangle$. We distinguish the following two cases: (1) If fd contains only input VOs on its right-hand side, i.e., $i = 0 \,\forall\, \langle w, b, i \rangle \in DEP_1$, add $sum(fd)$ to EFDM. If EFDM already contains a EFD for the left-hand side of $sum(fd)$, this EFD has to be overridden. (2) If fd contains a local VO $vo = \langle w, b, i \rangle$ on its right-hand side, i.e., $i \neq 0$, then EFDM has to contain an EFD $efd = \langle w, DEP_2 \rangle$. Compute $sum(fd)$ and substitute all variables w by DEP_2. Again, add the new EFD to EFDM and, if necessary, override an existing EFDM.
- After all FDs have been processed $EFDM = sum(FDM)$ holds.

Like internal models external models can be visualized by the use of digraphs. In contrast to digraphs of internal models, now variables (and not variable occurrences) are the nodes of the graph connected by edges representing EFDs. As already mentioned, graphs of external methods may contain cycles.

Example 6.6.2 *Let us, again, look at the source code fragments depicted in Figures 6.2 and 6.3. The digraphs resulting from the blocks' external FDMs are depicted in Figures 6.6 and 6.7.*

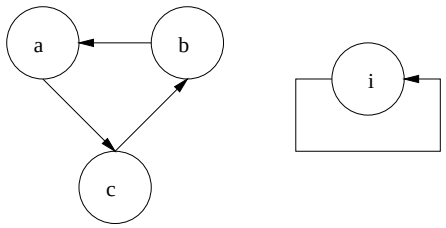


Figure 6.7: External model of block b_2

6.7 Combining FDMs

In this section we discuss the possibility of combining multiple FDMs to a single FDM. Imagine, for instance, two FD method models of the methods m and n, i.e., FDM_m and FDM_n. We then might be interested in all FDs arising from either m or n and collect them in a combined model $FDM_{m,n}$. Note that the same question can be asked in the context of expression, statement, or block models.

It is obvious that it does not make sense to combine two internal FDMs. The main reason is that internal FDMs consist of FDs, which are computed for VOs. As different methods also have different VOs a combination of both models is not feasible.

This, however, is not the case with external, i.e., summarized, FDMs. Summarized FDMs contain EFDs describing all dependencies of a certain variable. If we assume that a variable v is altered in both methods, then both models include exactly one EFD with v on its left-hand side. By combining these two EFDs we get a new EFD, which is part of the combined FDM, i.e., $FDM_{m,n}$. More formally, we state:

Definition 6.7.1 *Let* $fd_1 = \langle vo, DEP_1 \rangle$ *and* $fd_2 = \langle vo, DEP_2 \rangle$ *be two functional dependencies defined for the same variable occurrence vo. Then* $comb(fd_1, fd_2) = \langle vo, DEP \rangle$ *with* $DEP = DEP_1 \cup DEP_2$.

Definition 6.7.2 *Let* FDM_1 *and* FDM_2 *be two FDMs. The combined FDM, i.e.,* $FDM_{1,2}$ *is defined as* $FDM_{1,2} = comb(sum(FDM_1), sum(FDM_2))$, *where* $FDM_{1,2}$ *contains the following FDs:*

- $fd_1 = \langle v, DEP_1 \rangle$ *if* $fd_1 \in sum(FDM_1)$ *and* $\nexists fd_2 = \langle v, DEP_2 \rangle \in sum(FDM_2)$
- $fd_2 = \langle v, DEP_2 \rangle$ *if* $fd_2 \in sum(FDM_2)$ *and* $\nexists fd_1 = \langle v, DEP_1 \rangle \in sum(FDM_1)$
- $comb(fd_1, fd_2)$ *with* $fd_1 = \langle v, DEP_1 \rangle$ *and* $fd_2 = \langle v, DEP_2 \rangle$ *if* $fd_1 \in sum(FDM_1)$ *and* $fd_2 \in sum(FDM_2)$.

As already mentioned combined models can be computed for all sorts of external models (expression, statement, method models, etc...). In the following

chapters we will need combined models, for instance, to combine models for multiple evaluation traces (see Chapter 7) and to model method calls (see Chapter 9).

6.8 Properties of FDMs

Finally, we discuss some general properties, which are valid for all FDMs.

- A FDM is said to be *sound* or *correct*, if it only computes FDs that possibly (in the static case) or definitely (in the dynamic case) arise during run-time. The difference between a purely static and a dynamic approach lies in the modeling of potential influences as discussed in Section 6.3. However, a model containing FDs which cannot arise at run-time is not regarded as a sound model.

- A FDM is said to be *complete*, if it contains all FDs that are necessary to model a certain program run (dynamic case) or all possible program runs (static case). Again, it cannot be regarded as complete, if a FD arising from the used evaluation trace or one arising in any possible evaluation trace is missing.

- A FDM is said to be *minimal*, if it does not contain any spurious FDs. A model containing incorrect FDs therefore can never be minimal. A model containing potential dependencies is regarded as minimal in respect to a static analysis of the source code, but does not have to be minimal in respect to a single evaluation trace.

- Given an evaluation trace, all run-time information needed is available. This means that in principle all data and control dependencies can be computed using the trace and the `Java` programming language semantics.

- In the static case in general no minimal FDM with respect to a single trace can be computed, because of unknown branching decisions at run-time.

In the following sections we present in detail how three types of FDMs are created. The first model, which makes use of a concrete evaluation trace of a particular method, can be shown to be complete and minimal. However, these models have certain drawbacks, which do not always make them ideal for debugging (see Chapter 7). We therefore present a second model, which is computed statically without using knowledge about concrete variable values. This model is an approximation of the ideal model. It can be shown to be complete, but in general it is not correct and minimal (see Chapter 8). Finally, we present a simplification of these two models, which is based on a simplified and more abstract representation. We discuss its benefits and problems along with an altered interpretation in Chapter 14.

Chapter 7

The *ETFDM*

The first model type we present in this work is the so called Evaluation Trace Functional Dependency Model (*ETFDM*). During the computation of the *ETFDM* we make use of an evaluation trace (ET) and thus compute a model, which does not model a method m in general, but in the context of a particular evaluation trace. As defined in Chapter 6, a FDM consists of a set of FDs. We first describe how FDs in *ETFDMs* look like and how they differ from the general description given in Chapter 6. We then show how a concrete FDM for a given ET can be constructed. Finally, we discuss how multiple *ETFDMs* can be combined to one model and ultimately a FDM comprising all possible ETs.

7.1 FDs in *ETFDMs*

In Chapter 6 we have already defined the general format of a FD, i.e., $\langle vo, DEP \rangle$ with $vo = \langle v, m, i \rangle$. In the context of the *ETFDM* we use the same format of FDs, but define an extended DEP structure on the FDs' right-hand sides. This is done in order to handle not only influences of VOs, but also dependencies on run-time values, class instances, and method declarations. The constituents of a DEP structure are:

$R \subseteq VALUES$ contains run-time values, e.g., 10, that influence the FD's left hand side. These values can either be constants taken directly from the source code or values computed by the run-time system in the course of changing the value of v.

$V \subseteq VO$ contains variable occurrences influencing the FD's left hand side. $vo_i \in V$ can have the same form as vo. It can thus be of primitive or of reference type.

$M \subseteq METHODS$ contains method and constructor declarations that influence the value of the left-hand side's variable v. This, for instance, occurs in the case of a method call.

$O \subseteq OBJECTS$ contains the locations of all objects, not references to these locations. A location $o \in O$ on the FD's right-hand side means that v

references the object at location o. v is therefore said to be dependent on location o.

In the context of the *ETFDM* we write $DEP = \langle R, V, M, O \rangle$. Despite the extended format of a DEP structure, all definitions and algorithms proposed in Chapter 6 can still be applied to *ETFDMs* by analogy. In particular, we state:

Definition 7.1.1 (Equivalence (DEP)) *Let* $DEP_1 = \langle R_1, V_1, M_1, O_1 \rangle$ *and* $DEP_2 = \langle R_2, V_2, M_2, O_2 \rangle$ *be two DEP structures.* $DEP_1 = DEP_2$ *holds iff* $R_1 = R_2 \wedge V_1 = V_2 \wedge M_1 = M_2 \wedge O_1 = O_2$.

Note that Algorithm 6.6.2 for computing summarized FDMs given in Chapter 6 can still be applied to the new format of the DEP structure, if the sets R, M, and O are regarded as static and simply copied to the EFDs of the summarized model.

Let us now have a look at a FD in the context of the *ETFDM*. The format of a FD in the *ETFDM* is the same as described in Chapter 6, i.e., $\langle vo, DEP \rangle$ with DEP being of the form introduced above. The VO $vo = \langle v, b, i \rangle$ is the left-hand side of the FD and stands for the variable whose value is modified in a particular statement of block b. Variable v therefore depends on the FD's right-hand side. A change in the right-hand side of the FD may cause a change in the value of v. v can be of primitive or of reference type. If v is of primitive type (e.g. of type integer) a change in the FD's right-hand side alters the value of v. If on the other hand v is of reference type, a change in the FD's right-hand side changes the reference v. In the latter case v then references a different location, i.e., object, whereas the object stored at that particular location is still unaltered. v can be of the following form:

***x*:** in this case x denotes a local variable of the currently modeled block b

***0.x*:** here x stands for an instance field of the class containing the currently modeled method

***n.x*:** in this case n refers to the location of an object different from 0. v denotes an instance field of this particular object.

***a.x*:** in this case x is a static variable and a denotes the fully qualified name of the class, in which x is defined.

Note that in all three cases v can be of primitive or of reference type.

This shows us that when computing the *ETFDM* of a method m we get all FDs arising during run-time. The most important point is that locations represent objects as they are created at run-time. The numbers and types of created locations should exactly match the numbers and types of all instances created for the given ET. With the *ETFDM* the relationship between a VO on a FD's left-hand side and the associated DEP structure is always defined in a way that a change of the right-hand side *does* influence the left-hand side and in most cases results in a change of the value on the left-hand side. The reason

```
    int etfdm1(int x, int y) {
      int a,b,c;
1.    a = 3*x*x;
2.    b = 2*y*y;
3.    c = -5*x*y;
4.    return (a+b+c);
    }
```

Figure 7.1: Example method *etfdm1(int x, int y)*

for that is that only data and control dependencies as they occur during runtime are modeled. Potential influences (see Section 6.3) stemming from unknown branching or looping decisions are eliminated in the *ETFDM* by the use of ETs.

> ***JADE:*** *The* ***JADE*** *modeling component creates ETFDMs which contain FDs matching the format described above. The only difference is that instead of all run-time values on the right-hand side only constants (taken directly from the source code) are used. All run-time values could theoretically be taken from the used ET, but are substituted by constants in the source code. This, however, is no problem for two reasons: (1) Currently, only variable dependencies are used for debugging and (2) As we are only interested in source code bugs and assume that the run-time system works correctly, the modeling of run-time values is not actually needed. Note that in the following sections all stated ETFDMs only contain constants instead of run-time values for the sake of simplicity.*

7.2 Collecting the FDs of a single evaluation trace

We can now construct an *ETFDM* of a given expression, statement, or method by collecting all FDs arising from an ET. Assume we want to create a model of method m, i.e., FDM_m. This can be done by computing the models of all statements of m one after another. For each statement s and all sub-expressions and sub-branches a concrete ET has to be present. More formally, we define:

Definition 7.2.1 *The ETFDM of method m and a given ET, et, is defined as* $ETFDM_m^{et} = \{ETFDM_s^{et} \mid s \in m\}$.

Note that the computation of expression and statement models will be described in detail in Chapters 9 and 10.

Example 7.2.1 *Consider a very simple method etfdm1(int x, int y) computing the function* $t = 3x^2 + 2y^2 - 5xy$ *for its two parameters x and y. The source code of method etfdm1(int x, int y) is given in Figure 7.1. Before we can model*

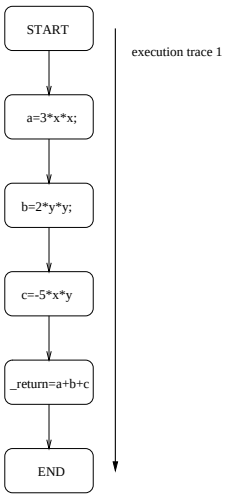


Figure 7.2: Evaluation trace of method *etfdm1(int x, int y)*

method etfdm1(int x, int y) we need an arbitrary ET for which we compute our model. Since there are no selection and loop statements in method etfdm1(int x, int y), the control flow of all ETs for etfdm1(int x, int y) looks like the one depicted in Figure 7.2. Therefore, we model exactly the same statements and (if ignoring all run-time values) get the same FDMs for all ETs. The ETFDM of method etfdm1(int x, int y) reads as follows:

$$\mathbf{ETFDM}^{(\mathbf{1},\mathbf{1})}_{\textit{etfdm1(int x, int y)}}$$

$$St.1: \; a_1 \leftarrow \{\{3\},\{x_0\},\{\},\{\}\}$$

$$St.2: \; b_1 \leftarrow \{\{2\},\{y_0\},\{\},\{\}\}$$

$$St.3: \; c_1 \leftarrow \{\{-5\},\{x_0,y_0\},\{\},\{\}\}$$

$$St.4: \; \_return \leftarrow \{\{\},\{a_1,b_1,c_1\},\{\},\{\}\}$$

Example 7.2.2 *Now consider another method etfdm2(int x, int y, int z), which models the following function:* $t = 3x^2 + 2y^2 - 5x$ *if* $z = 0$ *and* $t = 3x^2 + 2y^2 + xy$ *else. The source code of method etfdm2(int x, int y, int z) is given in Figure 7.3. If we look at the control flow chart of method etfdm2(int x, int y, in z) (see Figure 7.4) we see that there exist two different types of ETs depending on the evaluation value of the condition in line 3. If we take an input vector* $(1,2,0)$ *the then-block at line 3.1. is evaluated, whereas with an input of* $(1,2,3)$ *the else-block (line 3.2.) is executed. For the two cases we get two different ETFDMs , which are denoted as follows:*

```
        int etfdm2(int x, int y, int z) {
          int a,b,c;
1.        a = 3*x*x;
2.        b = 2*y*y;
3.        if (z == 0) {
3.1.        c = -5*x; }
          else {
3.2.        c = x*y; }
4.        return (a+b+c);
        }
```

Figure 7.3: Example method *etfdm2(int x, int y, int z)*

$\mathbf{ETFDM}^{(\mathbf{1},\mathbf{2},\mathbf{0})}_{\textit{etfdm2(int x, int y, int z)}}$

$St.1: \; a_1 \leftarrow \{\{3\},\{x_0\},\{\},\{\}\}$

$St.2: \; b_1 \leftarrow \{\{2\},\{y_0\},\{\},\{\}\}$

$St.3: \; c_1 \leftarrow \{\{-5\},\{x_0,z_0\},\{\},\{\}\}$

$St.4: \; \_return \leftarrow \{\{\},\{a_1,b_1,c_1\},\{\},\{\}\}$

$\mathbf{ETFDM}^{(\mathbf{1},\mathbf{2},\mathbf{3})}_{\textit{etfdm2(int x, int y, int z)}}$

$St.1: \; a_1 \leftarrow \{\{3\},\{x_0\},\{\},\{\}\}$

$St.2: \; b_1 \leftarrow \{\{2\},\{y_0\},\{\},\{\}\}$

$St.3: \; c_1 \leftarrow \{\{\},\{x_0,y_0,z_0\},\{\},\{\}\}$

$St.4: \; \_return \leftarrow \{\{\},\{a_1,b_1,c_1\},\{\},\{\}\}$

Note that there exists an infinite number of evaluation traces for both methods, *etfdm1(int x, int y)* and *etfdm2(int x, int y, int z)*. However, there exists only one possible control flow path for *etfdm1(int x, int y)* and two possible control flow paths for method *etfdm2(int x, int y, int z)*. Whereas R on the FDs' right-hand sides always depends on the concrete ET, all variable dependencies within one control flow path are the same. Therefore, if we only look at variable dependencies, the one FDM (two FDMs, respectively) described above cover all possible run-time scenarios of the two example methods.

7.3 Combining multiple ETFDMs

We now combine multiple *ETFDMs* of a certain method in order to create a model covering all FDs arising during different method executions. Let t_1 and t_2 be two evaluation traces of a given method m and $ETFDM_m^{t_1}$ and $ETFDM_m^{t_2}$ the

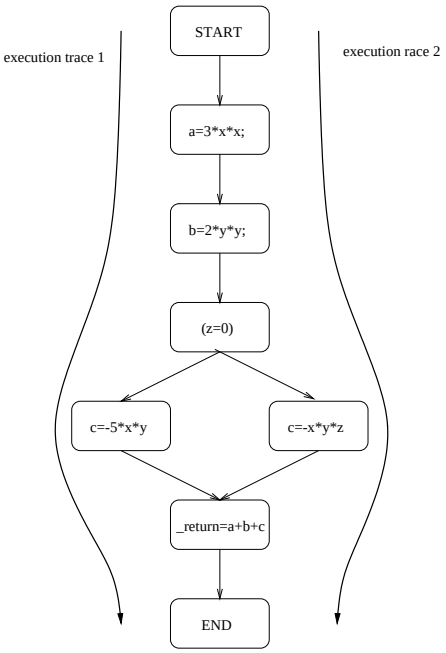


Figure 7.4: Evaluation trace of method *etfdm2(int x, int y, int z)*

ETFDMs computed for t_1 and t_2, respectively. We now construct an *ETFDM* which contains all FDs arising in either t_1 or t_2. Unfortunately, as shown in Chapter 6, the combination of two internal FDMs is not feasible. We can only compute a combined external FDM covering both evaluation traces. Formally, we write:

Definition 7.3.1 *The combined (external) ETFDM for the traces t_1 and t_2 is defined as: $sum(ETFDM_m^{t_1,t_2}) = comb(sum(ETFDM_m^{t_1}), sum(ETFDM_m^{t_2}))$.*

If we need an internal FDM combining $ETFDM_m^{t_1}$ and $ETFDM_m^{t_2}$, i.e., $ETFDM_m^{t_1,t_2}$, we make use of a little workaround. Depending on the smallest level of granularity needed, we can summarize all components of both *ETFDMs* and combine them to new components of the resulting combined model. If, for instance, we define the statement level as the smallest possible modeling level, we can compute summarized statement models for all statements $s \in m$ for both models and combine them to new statement models. The combined statement models will then constitute the combined *ETFDM* $ETFDM_m^{t_1,t_2}$. More formally, we write:

Definition 7.3.2 *Let $ETFDM_m^{t_1}$ and $ETFDM_m^{t_2}$ be two ETFDMs of method m for the evaluation traces t_1 and t_2, respectively. The combined ETFDM $ETFDM_m^{t_1,t_2}$ is defined as follows: $ETFDM_m^{t_1,t_2} = \{ETFDM_s^{t_1,t_2} \mid s \in m \wedge ETFDM_s^{t_1,t_2} = comb(sum(ETFDM_s^{t_1}), sum(ETFDM_s^{t_2}))\}$.*

Example 7.3.1 *Let us now come back to example method etfdm2(int x, int*

y, int z). The combination of the two ETFDMs as computed above, i.e., $ETFDM^{(1,2,0),(1,2,3)}_{etfdm2(int\ x,\ int\ y,\ int\ z)}$ *reads as follows:*

$\mathbf{ETFDM}^{(\mathbf{1},\mathbf{2},\mathbf{0}),(\mathbf{1},\mathbf{2},\mathbf{3})}_{\textbf{\textit{etfdm2(int x, int y, int z)}}}$

$St.1: \ a_1 \leftarrow \{\{3\}, \{x_0\}, \{\}, \{\}\}$

$St.2: \ b_1 \leftarrow \{\{2\}, \{y_0\}, \{\}, \{\}\}$

$St.3: \ c_1 \leftarrow \{\{-5\}, \{x_0, y_0, z_0\}, \{\}, \{\}\}$

$St.4: \ \_return \leftarrow \{\{\}, \{a_1, b_1, c_1\}, \{\}, \{\}\}$

Note that it is also possible to combine the summarized FDMs for each expression $e \in m$. We then get a more detailed model, but have to face the higher computational requirements of such a task.

7.4 A complete FDM

Ultimately, if we want to create a FDM of method m, i.e., FDM_m, which covers all possible run-time scenarios, we can compute the FDM of all possible evaluation traces of a certain method and combine them to a general model. This approach is briefly described in this section.

We begin with defining the set of all possible evaluation traces through a method m. This set will usually be very large and in most cases include an infinite number of evaluation traces. Formally, we define:

Definition 7.4.1 *ET_m is defined as the set of all possible evaluation traces through method $m \in METHODS$.*

We now define the complete functional dependency model (*CFDM*) of method m by combining the *ETFDMs* of all possible evaluation traces. Formally, we say:

Definition 7.4.2 $CFDM_m = comb(ETFDM^{t_1}_m, ..., ETFDM^{t_n}_m) \mid t_1, ..., t_n \in ET_m$.

Unfortunately the *CFDM* of a given method cannot be computed in the general case, because in most cases there exists an infinite number of evaluation traces for m, i.e., $|ET_m| = \infty$. Even if the number is finite, the computational requirements of such a task would exceed satisfying modeling times by far.

Example 7.4.1 *If we come back to method etfdm1(int x, int y) we find that an infinite number of evaluation traces exists, which prevents us from computing* $CFDM_{etfdm1(int\ x,\ int\ y)}$. *Nevertheless, there is only one control flow path through method etfdm1(int x, int y). This means that the ETFDM shown above covers all variable dependencies as each evaluation trace in the same control flow path produces the same variable influences.*

Example 7.4.2 *If we look at method etfdm2(int x, int y, int z) we, again, encounter an infinite number of evaluation traces. However, there are only two control flow paths through the method as depicted in Figure 7.4. This means that the above combined ETFDM covers all variable dependencies possibly created at run-time.*

7.5 Handling aliasing with the *ETFDM*

As an object-oriented programming language, `Java` features the concept of aliasing. The basis of this phenomenon is the explicit distinction between class instances (locations) and variables of reference type (references), which is inherent to all object-oriented systems. A location l may be referenced by more than one variables of reference type. Whereas a change of one variable v does not alter the content of l, a modification of l on the other hand has no effect on variable v. In this work we use the following definition:

Definition 7.5.1 *Two variables of reference type, v and w, which reference the same run-time location l, are said to cause aliasing. During the modeling process we face an aliasing problem, if the content of l is altered by accessing l through one of its references, say v. A modification of l should then be visible in the model not only for an access of l through v, but also for an access of l through w.*

The *ETFDM* handles aliasing in its most straightforward manner by keeping locations and variables separate. This approach obviously solves the aliasing problem during the modeling process, because changes of references and modifications of locations are represented by different model components, i.e., variable names including the location of their owner instances on the one hand and locations on the other.

Example 7.5.1 *Let us come back to the example presented in Section 5.2 (see Figure 5.1). We now add a new method aliasing() (see Figure 7.5) to the* `Java` *system, which creates one instance of class Point being referenced by two distinct variables of reference type, i.e., p1 and p2. Following the above definition, we now have a classic example of aliasing. By calling method doubleXValue() in statement line 3 the content of the point object is altered leaving the reference p1 unchanged. The field access in line 4 should therefore return the new value of field x, no matter whether the point object is accessed via variable p1 or p2.*

If we now look at the resulting ETFDM we see that the created location, i.e., location 1, and its instance fields 1.x and 1.y are kept separate from the reference variables p1 and p2. So, the ETFDM of line three includes a FD for 1.x, but none for either p1 or p2.

```
      void aliasing() {
        Point p1, p2;
1.      p1 = new Point(0,0);
2.      p2 = p1;
3.      p1.doubleXValue();
4.      int tmp = p2.x;
      }
```

Figure 7.5: Example method *aliasing()*

$\mathbf{ETFDM}^{()}_{aliasing}$

$St.1: \ 1.x_1 \leftarrow \{\{0\},\{\},\{Point(int,int)\},\{\}\}$

$St.1: \ 1.y_1 \leftarrow \{\{0\},\{\},\{Point(int,int)\},\{\}\}$

$St.1: \ p1_1 \leftarrow \{\{\},\{\},\{Point(int,int)\},\{1\}\}$

$St.2: \ p2_1 \leftarrow \{\{\},\{p1_1\},\{\},\{1\}\}$

$St.3: \ 1.x_2 \leftarrow \{\{2\},\{p1_1, 1.x_1\},\{doubleXValue()\},\{\}\}$

$St.4: \ tmp_1 \leftarrow \{\{\},\{p2_1, 1.x_2\},\{\},\{\}\}$

Note that in the above model only constants are used. Whereas 0 appears in line 1 of method aliasing(), the constant 2 is imported from method doubleX-Value(). As already explained, run-time values are not depicted in the ETFDMs herein for the sake of simplicity. Currently, they are not computed by the ***JADE*** *system.*

As shown during the computation of the *ETFDM* the aliasing problem is solved by representing a `Java` run-time system by variables on the one hand and locations on the other. Thus locations are crucial to the correct modeling of object-oriented systems.

7.6 Properties of the *ETFDM*

We conclude this chapter with a short discussion about the main properties of *ETFDMs*:

- All *ETFDMs* are sound models, because their FDs are directly taken from an evaluation trace and therefore can easily be shown to indeed arise at run-time.
- An *ETFDM* computed for a single evaluation trace *et* is both, complete and minimal, in relation to *et*. In other words, if making use of a single evaluation trace, we get all FDs arising from this particular program run, but no additional FDs stemming from other run-time scenarios. However,

such a model cannot be complete in relation to the *CFDM* covering all run-time cases.

- A combined *ETFDM*, i.e., an *ETFDM* computed for two or more evaluation traces, is no longer minimal in relation to a single program run. Again, it is not complete in relation to a complete model.

- A complete *ETFDM* covering all run-time scenarios, i.e., *CFDM*, is complete, because it contains all FDs, which might arise in any of all possible program runs. Nevertheless, it is not minimal in relation to a single evaluation trace.

- The *ETFDM* covers all data and control dependencies arising at run-time. Note that potential influences (see Section 6.3) are not part of the model.

- All constituents of the *ETFDM*, i.e., run-time values, objects, and variables are taken from an evaluation trace. The model is therefore detailed and easy to interpret.

- An *ETFDM* can only be computed for a particular run of a given method. It does not make sense to compute a whole host environment model. The drawback of this approach is that the models of all sub-blocks and called methods are different (due to different sub-traces) and have (in the case of hierarchical debugging) to be stored separately. This may require a lot of memory.

- Multiple *ETFDMs* can be combined to a model covering two or more evaluation traces. The combination of *ETFDMs* might result in a *CFDM* including all FDs, which arise in any of all possible program runs. However, in the general case the set of all possible evaluation traces is either unknown or infinite.

- The time needed to compute the *ETFDM* for a single evaluation trace is proportional to the time needed to execute the program. This seems obvious, if we consider that each statement of the method to be modeled has to be modeled only once. In most cases *ETFDMs* can be computed in a reasonable time. However, the computation of a *CFDM*, if possible, has huge computational requirements and in most cases is not possible.

- If a model for a single program run, i.e., test-case, is needed, the *ETFDM* provides an appropriate model, which is sound, complete, and minimal in relation to this particular evaluation trace. The combination of multiple *ETFDMs* cannot be regarded as a useful approach towards the creation of a general program model covering all run-time scenarios. In the following chapter we discuss another model type, which approximates the *CFDM* in the general case.

Chapter 8

The *DFDM*

In this section we describe a second class of FDMs, the Detailed Functional Dependency Models (*DFDMs*). A *DFDM* tries to model all FDs of a certain method m without the use of evaluation traces or other run-time information. The creation of a *DFDM* is a purely static approach, which only relies on the source code of the analyzed method.

As we will see there are certain language features, which cannot be modeled exactly unless run-time information is present. In many cases the *DFDM* therefore represents only an approximation of the *CFDMs* described in Chapter 7, which sometimes is achieved through a higher level of abstraction compared with the *ETFDM* described in Chapter 7.

We first describe the meaning of the different parts of a FD in *DFDMs*, which somehow differ from their counterparts in *ETFDMs*. We then show how a *DFDM* can automatically be constructed using only a method's source code. Finally, we discuss some of the differences to the *ETFDM* and specific properties of the *DFDM*.

8.1 FDs in the *DFDM*

As with *ETFDMs* we start with a short analysis of the individual parts of FDs as they constitute a *DFDM*. Again, the form of FDs is based on the general format of a FD, i.e., $\langle vo, DEP \rangle$ with $vo = \langle v, m, i \rangle$, as it was presented in Chapter 6. As with *ETFDM* we use an extended form of the DEP structure in the context of *DFDMs*, which in the context of the *DFDM* includes all language constructs that possibly influence the FD's left-hand side during run-time. DEP itself consists of the following elements:

$C \subseteq CONSTS$ contains all constant values, e.g., 10, that possibly influence the FD's left-hand side.

$V \subseteq VO$ contains all variable occurrences possibly influencing the FD's left-hand side.

$M \subseteq METHODS$ contains all method and constructor declarations that possibly influence the value of the left-hand side's variable v. This, for instance,

occurs in the case of a method call.

$L \subseteq LOCATIONS$ contains locations of objects, not references to these locations. A location $l \in L$ on the FD's right-hand side means that v is possibly referencing the object at location l. v is therefore said to be dependent on location l.

In the context of the *DFDM* we write $DEP = \langle C, V, M, L \rangle$. Despite the extended format of a DEP structure, all definitions and algorithms proposed in Chapter 6 can still be applied to *DFDMs* by analogy. In particular, we state:

Definition 8.1.1 (Equivalence (DEP)) *Let* $DEP_1 = \langle C_1, V_1, M_1, L_1 \rangle$ *and* $DEP_2 = \langle C_2, V_2, M_2, L_2 \rangle$ *be two DEP structures.* $DEP_1 = DEP_2$ *holds iff* $C_1 = C_2 \wedge V_1 = V_2 \wedge M_1 = M_2 \wedge L_1 = L_2$.

Note that Algorithm 6.6.2 for computing summarized FDMs given in Chapter 6 can still be applied to the new format of the DEP structure, if the sets C, M, and L are regarded as static and simply copied to the EFDs of the summarized model.

Let us now, again, have a look at a FD in the context of the *DFDM*. Generally, the format of a FD in the *DFDM* is the same as described in Chapter 6, i.e., $\langle vo, DEP \rangle$ with DEP being of the form introduced above. The VO $vo = \langle v, b, i \rangle$ is the left-hand side of the FD and stands for the variable whose value is modified in a particular statement of block b. In contrast to *ETFDMs* a change in the right-hand side of the FD does not necessarily cause a change in the value of v. It is possibly influenced by the constituents on the right-hand side. Again, v can be of primitive or of reference type. If v is of primitive type (e.g., of type **int**), a change in the FD's right-hand side may alter the value of v. On the other hand, if v is of reference type, a change in the FD's right-hand side may change the reference v. The format of v does not differ from the one in *ETFDMs*.

Note that this interpretation of a FD is no longer identical with the interpretations of FDs used in the context of FDMs in general (see Chapter 6) or *ETFDMs* (see Chapter 7). The most important peculiarities of the *DFDM* are:

- In contrast to *ETFDMs* only constants and literals appearing in the source code are used on the FDs' right-hand sides. Run-time values are not available during a purely static analysis and therefore are not part of the model. We only use variable dependencies for the debugging of `Java` source files (see part III). Therefore, missing run-time values will not affect the performance of the debugging tool.

- In contrast to the *ETFDM*, in the *DFDM* locations no longer represent objects created at run-time, but class instances which *might* be created at run-time. The number of the created locations does not always match the number of all instances created at run-time, because the *DFDM* tries to cover all run-time scenarios and not just one particular program run. Therefore, a FD's right-hand side might include locations, which are not created in one particular program run of method m.

- The most important difference to all previously described models is that a change to a FD's right-hand side *may*, but not necessarily does change the value of the FD's left-hand side. Once more, the reason is that we are employing a purely static approach, which due to a lack of run-time information does not only have to consider data and control dependencies as they occur during run-time, but also all potential influences (see Section 6.3) stemming from unknown branching or loop decisions.

8.2 Creating a *DFDM*

In Chapter 7 we theoretically describe *CFDMs*, which come into being through the combination of multiple *ETFDMs*. We argue that if combining *ETFDMs* for all possible ETs, we get an optimal FDM covering all ETs and run-time scenarios. As shown this approach has two main drawbacks: (1) All possible ETs or at least all possible paths through a method m have to be known and (2) even then the computational complexity of such a task is enormous in most cases. The main goal of the *DFDM* is to approximate a *CFDM*. By computing only one model and ignoring any run-time information, the drawbacks of the approach mentioned above are thus avoided. Unfortunately, due to a lack of run-time information in some cases the *CFDM* can only be an approximation and computes more FDs as there possibly arise during run-time. The following examples demonstrate problems in the context of *DFDMs*:

Example 8.2.1 *In some cases the FDs can be too large. Imagine, for instance, the following source code fragment:*

```
      int x;
1.    if (false)
1.1.    x=3;
      else
1.2.    x=2;
```

Since the condition in line 1 always evaluates to `false`*, the statement in line 1.1 is never executed. The FD* $x_1 \leftarrow \{3\}$ *thus can not be part of any ETFDM of statement line 1. Clearly, if we want to construct a CFDM covering all run-time scenarios, we can ignore statement line 1.1, because no run-time FDs can arise from it. When computing the DFDM of such a method, we can not rely on any run-time information and have to add the FD arising from statement line 1.1 to our model. This leads to the FD* $x_1 \leftarrow \{2, 3\}$*, which includes a superfluous element, i.e.,* 3*. Note that in this simple example the evaluation value of the condition in line 1 could easily be determined at compile-time. However, this is not possible in the general case.*

Example 8.2.2 *It is possible that there are too many FDs created in a DFDM. Consider the following source code fragment:*

```
      int x;
1.    x=2;
2.    if (false)
2.1.    x=3;
```

In this case an additional FD is created for the selection statement in line 2. However, this FD would not be necessary, because the branch in line 2.1 will never be executed at run-time.

The hardest problems during the creation of a *DFDM* are posed by loop statements and recursive method calls, which create a finite but unknown number of locations at run-time (as long as the program terminates). The exact modeling of these program structures and other statements and expressions is described in Chapters 9 to 13.

8.3 Handling aliasing with the *DFDM*

When computing the *DFDM* of a given method, in principle we are facing the same aliasing problems as with the *ETFDM*. However, the basic model components of the *DFDM* are the same as the ones used for the *ETFDM*. Consequently, aliasing problems can be solved just like with the *ETFDM*, i.e., by keeping locations and reference variables separate (see Section 7.5 for a detailed elaboration of this topic). Note that the *DFDM* for method *aliasing()* (see Figure 7.5) is the same as the *ETFDM* presented in Section 7.5.

The *DFDM* might include a higher number of locations due to the modeling of potential dependencies, which are not considered in the *ETFDM*, where an evaluation trace is available. However, this does not change the basic modeling strategy or the properties of the model components.

8.4 Properties of the *DFDM*

The *DFDM* of a given method is supposed to cover all FDs arising at run-time by applying a purely static analysis of the method's source code. The following properties of the *DFDM* can be stated:

- In contrast to *ETFDMs DFDMs* are static models. Their creation does not involve any information about a particular evaluation trace or program run. The only information used during the modeling process is the source code of the `Java` system and the programming language semantics as defined in [16].
- A *DFDM* approximates a *CFDM* by covering all run-time scenarios, which seem possible at compile-time.
- When computing a *DFDM*, each method is modeled only once. This leads to (1) a faster modeling process, (2) reusable models of all methods, (3) less

computational requirements than for the *ETFDM*, and (4) the creation of models for whole host environments.

- *DFDMs* are not always sound as they might compute FDs, which will never arise in any run-time scenario. Therefore, the *DFDM* can only be an approximation of the *CFDM*.

- A *DFDM* is complete, because it covers all possible FDs. In the context of locations, this proposition is true, only if certain abstractions are introduced into the model. These abstractions are in detail described in the following chapters. Note that without these abstractions the *DFDM* of an arbitrary method cannot always be computed, because of an infinite set of possible FDs (arrays, loops, recursion). By introducing a more abstract model semantics, we are able to compute *DFDMs* in the general case with the drawback of getting a less detailed and precise model in comparison with the *ETFDM*.

- A *DFDM* is not minimal, neither in relation to a single evaluation trace, nor in relation to the *CFDM*. This is obvious, because FDs might be created, which are never produced at run-time.

- A *DFDM* comprises all types of FDs described in Section 6.3, i.e., data dependencies, control dependencies, and potential influences.

Chapter 9

Modeling Expressions

At expression level, FDs can only arise from variable assignments or method calls. In the latter case FDs originating from assignments to variables in the called method, which are visible in the calling method, are imported into the calling method. If we want to compute the FDM of an arbitrary expression, we have to traverse the expression's parse tree and collect all FDs arising from either assignment or method call nodes. This section is dedicated to the exact modeling of variable assignments and method call expressions. Note that some kinds of conditional expressions have to be considered separately, because of their inherent control dependencies. The modeling of these structures is shortly discussed at the end of this section and in more detail in Chapter 10.

9.1 Variable assignments

Strictly speaking, only variable assignments can be the source of data dependencies. Whereas FDs imported into an expression through a method or constructor call are ultimately based on variable assignments, FDs arising from selection and loop statements are merely control or potential dependencies.

The modeling of variable assignments is performed in a straightforward way. Whereas the assignment's target variable becomes the FD's left-hand side, the assignment's right-hand side is transformed into the new FD's DEP structure. This is done by adding all constants, variables and methods influencing the assignment's right-hand side to the new FD's DEP structure. In case of the *ETFDM* additional run-time information is also put to the DEP structure, such as run-time values. If the assignment's target variable is of reference type, all locations of the right-hand side have to be copied to the new DEP structure. In case of a class instance creation expression on the assignment's right-hand side the newly created location has also to be added to the DEP structure. Note that array creation and array initializer expressions are handled separately and are described in Chapter 11.

Note that if the left-hand side of the assignment is a field access, the scope of this field access has to be resolved before its member variable can be used as the FD's left-hand side. This, however, can be done non-ambiguously only in the case of *ETFDMs* when the exact location referenced by the scope of the field

```
     void assignment() {
        Point p1, p2, p3;
1.      p1 = new Point(0,0);
2.      p2 = new Point(1,1);
3.      p3 = p2;
4.      p3.x = 5
     }
```

Figure 9.1: Example method *assignment()*

access is known. When computing the *DFDM* we might get multiple locations being possibly referenced by the scope of the field access. In this scenario multiple FDs can arise from one variable assignment. Clearly, the model will be exacter and better suited for diagnosis when using evaluation traces together with the *ETFDM*.

Another interesting question is, whether reference variable occurrences in the scope of a variable assignment should be included in the DEP structure of the resulting FD. The reference on the assignment's left-hand side plays an important role in the assignment process, since an incorrect reference will cause the assignment's left-hand side to access an incorrect field, or to be more concise, the field of an incorrect object. In most cases this results in two objects being in an incorrect state, i.e., one, which is unintendedly altered, and one, which stays unaltered although it should have been modified. Note that the same applies to references in the scope of method calls.

Example 9.1.1 *Consider method assignment(), which is depicted in Figure 9.1. Method assignment() creates two locations of type Point, i.e., location 1 and 2. In statement line 3 the reference p3 is assigned to p2 and therefore references the same location as p2, i.e., location 2. Finally, in statement line 4 the field x of location 2 is modified through a field access via reference p3. Clearly, if 2.x is in an incorrect state after the execution of statement 4, the culprit can not only be statement line 4, but also statement line 3 and as a consequence statement line 2. The FD* $2.x \leftarrow p3$ *is therefore necessary in order to handle faults, which involve incorrect reference variables.*

Unfortunately, the approach described above introduces new problems, which can be demonstrated using method *test()* of class *Point* (see Figure 5.1). Assume we know that either the object created in statement line 1 or the object created in statement line 5 is in an incorrect state after the execution of *test()*. In both cases we get statement line 1 as potential diagnosis through the influence of *p1* on both, the assignments in lines 3 and 4 and the method call in line 5. Whereas in the case of the assignment statements we could argue that the field of the wrong object is altered and thus statement 1 is a diagnosis, this seems to be even more unrealistic in the case of the method call, because the content of the object referenced by *p1* is completely reset after statement 1. Therefore, if considering a reference in an assignment's or method call's scope in the resulting DEP structure, too many

diagnoses might be computed from the resulting model, making the debugging process suboptimal.

9.2 Method calls

The second way FDs may arise at expression level is through the call of methods or constructors. These FDs are herein called side-effect FDs, although they ultimately stem from variable assignments in the called method, just like assignments in the calling method. In the following we describe the modeling of method and constructor calls for both model types, the *ETFDM* and *DFDM*.

At compile-time a certain method call can easily be identified by its parse tree node *mc*. At run-time the situation is somewhat different, because we also have to know, which method actually gets called during the execution of *mc*. In a programming language like `Java` supporting polymorphism this question cannot always be answered at compile-time. Therefore, we give the following definition:

Definition 9.2.1 *A given method call at run-time can uniquely be identified by the tuple $\langle mc, md \rangle$, where mc stands for the parse tree node of the method call and md is the method declaration actually being called at run-time. We call the tuple $\langle mc, md \rangle$ a method call key.*

Generally, the following steps have to be performed, when modeling method or constructor calls:

1. Resolve the scope of the method call, i.e., determine the receiver(s) of the method call.
2. Find the declaration(s) of the called method(s).
3. Get the FDM(s) of the called method(s).
4. Transform the summarized FD method model(s) of the called method(s) to its (their) external representation(s) in the calling method.
5. If necessary, combine all summarized FDMs of the method call to a single (summarized) model.
6. Incorporate the summarized FDM of the method call into the FDM of the calling method.

These steps are discussed in the following sections in more detail. Throughout this section we use example methods creating instances of class *obj* and *obj2*. Class *obj* contains two instance fields, *value* and *field*, and, apart from its constructor, two methods, *getValue()* and *sideEffect(obj o)*. Whereas method *getValue()* simply returns the value of the instance field *value*, *sideEffect(obj o)* changes the instance field *field* through a side-effect. The source code of class *obj* is depicted in Figure 9.2. Class *obj2* is derived from class *obj* and overrides method *sideEffect(obj o)*. The overridden version of *sideEffect(obj o)* creates a new instance of class *obj* through a side-effect. The source code of class *obj2* is shown in Figure 9.3.

```
      class obj {
        int value;
        obj field;

        obj(int x) {
1.        value = x; }

        int getValue() {
1.        return value; }

        int sideEffect(obj o) {
1.        field = o;
2.        return value; }
      }
```

Figure 9.2: Example class *obj*

```
      class obj2 extends obj {
        obj2(int x) {
1.        super(x); }

        int sideEffect(obj o) {
1.        field = new obj(600);
2.        return value; }
      }
```

Figure 9.3: Example class *obj2*

9.2.1 Scope resolution

As a first step, the scope of the method or constructor call has to be resolved. This is necessary to determine the receiver of the method or constructor call. Generally, the scope of a method call can be empty, a fully qualified name, an arbitrary expression, or a keyword like **this** or **super**. At this stage it is important to know, whether the called method is a class or an instance method, and the exact class or instance, which is the receiver of the method call.

***ETFDM*:** When computing the *ETFDM*, the scope of a method call can always be resolved non-ambiguously, because the receiver of the method call can directly be retrieved from the evaluation trace. To be more precise, each variable of reference type should only reference one location at a time. FDs with two or more locations on their right-hand sides should not appear in *ETFDMs*.

For example, consider the `Java` method *mc1(int i)*, which is shown in Figure 9.4. Here, variable *o* is assigned two different objects (say, locations 1 and 2) in the two branches of an **if** statement. The source code of class *obj* is depicted in Figure 9.2.

```
      int mc1(int i) {
        obj o;
1.      if (i > 0) {
1.1.      o = new obj(100); }
        else {
1.2.      o = new obj(200); }
2.      return o.getValue();
      }
```

Figure 9.4: Example method *mc1(int i)*

If the parameter i is set to a value greater than 0, e.g., 1, statement 1.1 is executed and an instance of class *obj* with a value of 100 is created. This instance is also the receiver of the call to method *getValue()* in line 2. The *ETFDM* of method *mc1(int i)* reads as follows:

$$\mathbf{ETFDM}^{(1)}_{\mathbf{mc1(int\ i)}}$$

$$St.1: \ 1.value_1 \leftarrow \{\{0,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$$

$$St.1: \ o_1 \leftarrow \{\{0\},\{i_0\},\{obj.obj(int)\},\{1\}\}$$

$$St.2: \ \_result_1 \leftarrow \{\{\},\{1.value_1,o_1\},\{obj.getValue()\},\{\}\}$$

If, on the other hand, *mc1(int i)* is called with a value of $i \leq 0$, e.g., -1, statement 1.2 is executed. In this case another instance of class *obj* is created, this time with a value of 200. Again, this instance is the receiver of the call to method *getValue()* in line 2. The *ETFDM* of method *mc1(int i)* reads as follows:

$$\mathbf{ETFDM}^{(-1)}_{\mathbf{mc1(int\ i)}}$$

$$St.1: \ 1.value_1 \leftarrow \{\{0,200\},\{i_0\},\{obj.obj(int)\},\{\}\}$$

$$St.1: \ o_1 \leftarrow \{\{0\},\{i_0\},\{obj.obj(int)\},\{1\}\}$$

$$St.2: \ \_result_1 \leftarrow \{\{\},\{1.value_1,o_1\},\{obj.getValue()\},\{\}\}$$

Note that in both cases one instance of class *obj* is created. Both instances are given the index 1. Nevertheless, both instance are different objects and arise from different statements in the source code of method *mc1(int i)*.

***DFDM*:** In case of the *DFDM* the receiver of a given method call cannot always be resolved non-ambiguously. There exist FDs with multiple locations on their right-hand sides, mainly due to the modeling of selection statements (see Chapter 10).

If we look at method *mc1(int i)*, we find that after the **if** statement at statement line 1 variable o possibly references either location 1 or location 2. By a

```
      int mc2(int i) {
        obj o;
1.      obj o1 = new obj(1000);
2.      if (i > 0) {
2.1.      o = new obj(100); }
        else {
2.2.      o = new obj(200); }
3.      return o.sideEffect(o1);
      }
```

Figure 9.5: Example method *mc2(int i)*

purely static analysis (as the *DFDM*) this question cannot be decided. Therefore, the scope of the call to method *getValue()* cannot be resolved non-ambiguously. Consequently, the method call has to be modeled using all possible receivers, i.e., all locations which are currently possibly referenced by variable *o*. The *DFDM* of method *mc1(int i)* reads as follows:

$\mathbf{DFDM_{mc1(int\ i)}}$

$St.1: \ 1.value_1 \leftarrow \{\{0,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ 2.value_1 \leftarrow \{\{0,200\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ o_1 \leftarrow \{\{0\},\{i_0\},\{obj.obj(int)\},\{1,2\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\},\{o_1, 1.value_1, 2.value_1\},\{obj.getValue()\},\{\}\}$

We see that now two instances of class *obj* are created, i.e., locations 1 and 2. In statement line 2 both locations are possible receivers of the call to method *getValue()*, which leads to the return value of *mc1(int i)* being dependent on the instance field *value* of both locations.

We are now facing another problem, namely how to combine the multiple expression models we get for the call of method *getValue()*. In this example method *getValue()* is known to be side-effect free, which means that no side-effect FDs arise from the method call in statement line 2. Now consider a small modification of method *mc1(int i)*, which is depicted in Figure 9.5. This time *sideEffect(obj o)* is called, which through a side-effect changes the internal state of its receiver, i.e., the object referenced by variable *o*.

***ETFDM*:** When we compute the *ETFDM* of method *mc2(int i)*, the scope of variable *o* can be resolved non-ambiguously. Therefore, only one FDM of the method call in statement line 3 has to be computed. The following model of statement 3 shows that only one side-effect FD has been created.

$\mathbf{ETFDM}^{(1)}_{\mathbf{mc2(int\ i)}}$

$St.3:\ 2.field_1 \leftarrow \{\{\}, \{o_1, o1_1\}, \{obj.sideEffect(obj)\}, \{1\}\}$

$St.3:\ \_result_1 \leftarrow \{\{\}, \{o_1, 2.value_1\}, \{obj.sideEffect(obj)\}, \{\}\}$

***DFDM*:** When creating the *DFDM*, we have to compute two different FD models corresponding to the two possible receivers, location 2 and location 3. Note that here location 1 is the newly introduced object referenced by variable *o1*. As a result we get two different side-effect FDs. The FD of the return value contains both locations, location 2 and 3.

$\mathbf{DFDM}_{\mathbf{mc2(int\ i)}}$

$St.3:\ 2.field_1 \leftarrow \{\{\}, \{o_1, o1_1\}, \{obj.sideEffect(obj)\}, \{1\}\}$

$St.3:\ 3.field_1 \leftarrow \{\{\}, \{o_1, o1_1\}, \{obj.sideEffect(obj)\}, \{1\}\}$

$St.3:\ \_result_1 \leftarrow \{\{\}, \{o_1, 2.value_1, 3.value_1\}, \{obj.sideEffect(obj)\}, \{\}\}$

JADE: In case of ambiguous scope locations the JADE system computes all FD models of a method call one after another. Side-effect FDs are stored in an intermediate data structure, where they are accumulated until all models have been created. After that all side-effect FDs are written to the current FD method model.

Model comparison: Obviously, it is a big advantage of the *ETFDM* that less models of the called method have to be created. This not only makes the modeling process faster, but also results in fewer and smaller FDs in the calling method. However, it should be noted that the resulting FD method model is only correct for one specific evaluation trace, whereas the *DFDM* incorporates the FDs arising from all possible traces.

9.2.2 Method resolution

Once we know the receiver(s) of a method call we have to find the declaration of the called method. This includes matching the signature of the called method with all methods defined for the receiver and (in case of polymorphism) determine the method declaration actually being called at run-time. Of course this is not always possible, if we decide to do without evaluation traces.

***ETFDM*:** As with scope resolution the method being called at run-time can directly be taken from the evaluation trace. Even with polymorphism we can always determine a single method being called. This makes the modeling process

```
      int mc3(int i) {
        obj o;
1.      obj o1 = new obj(1000);
2.      if (i > 0) {
2.1.      o = new obj(100); }
        else {
2.2.      o = new obj2(200); }
3.      return o.sideEffect(o1);
      }
```

Figure 9.6: Example method *mc3(int i)*

an easy job as we always have to create exactly one FD method model when working with the *ETFDM*.

***DFDM*:** When we look at the creation of the *DFDM*, we find that statically we cannot always determine the method being called at run-time. We rather get a set of possibly called methods, which all have to be considered during the modeling process. To be more precise, we have to look at the type of the scope location, i.e., at the type of the currently modeled receiver. We can distinguish the following two cases:

1. If the location of the currently modeled receiver is a new location or an imported new location, the exact type of the receiver is known. In this case we can determine the method being called at run-time.

2. If, on the other hand, the location of the receiver is a default location or an imported default location, the receiver's run-time type is unknown, because the run-time type can be the type of the location or any sub-type derived from it. In the worst case we therefore have to create method models for all possibly called methods of all possible receivers. Obviously, this can result in a lot of models to be created and thus in a high number of FDs.

Example 9.2.1 *Method mc3(int i) (see Figure 9.6) is similar to method mc2(int i), but has slightly been modified in the* ***if*** *statement's else-branch. The created object is now of type obj2, which is derived from obj. Method sideEffect(obj o) has been overridden for class obj2. To distinguish it from method sideEffect(obj o) in class obj it creates an additional object of class obj. Note that the source code of class obj and obj2 is depicted in Figures 9.2 and 9.3, respectively.*

***ETFDM*:** When computing the *ETFDM* with an input value of $i > 0$, e.g., 1, we know that one object of type *obj* is created in statement line 2. We can therefore non-ambiguously resolve the method call in statement line 3 and find the correct method declaration to be modeled, i.e., the one defined in class *obj*. The FDs arising from statement 3 read as follows:

$\mathbf{ETFDM}^{(1)}_{\mathbf{mc3(int\ i)}}$

$St.3: \ 2.field_1 \leftarrow \{\{\},\{o_1,o1_1\},\{obj.sideEffect(obj)\},\{1\}\}$

$St.3: \ \_result_1 \leftarrow \{\{\},\{o_1,2.value_1\},\{obj.sideEffect(obj)\},\{\}\}$

If method *mc3(int i)* gets called with a parameter value of $i \leq 0$, e.g., -1, a new instance of class *obj2* is created. In statement line 3 method *sideEffect(obj o)* of *obj2* is called, which leads to the creation of an additional instance of *obj*, i.e., location 3. The *ETFDM* of statement line 3 reads as follows:

$\mathbf{ETFDM}^{(-1)}_{\mathbf{mc3(int\ i)}}$

$St.3: \ 2.field_1 \leftarrow \{\{\},\{o_1\},\{obj.obj(int),obj2.sideEffect(obj)\},\{3\}\}$

$St.3: \ 3.value_1 \leftarrow \{\{600\},\{o_1\},\{obj.obj(int),obj2.sideEffect(obj)\},\{\}\}$

$St.3: \ \_result_1 \leftarrow \{\{\},\{o_1,2.value_1\},\{obj2.sideEffect(obj)\},\{\}\}$

***DFDM*:** Statically the exact type of the object referenced by *o* cannot be determined. Here, the scope of the method call in statement line 3 cannot only reference two different locations, but also two locations of different type, i.e., *obj* and *obj2*. Again, two models have to be computed and combined to the following external model of statement line 3:

$\mathbf{DFDM}_{\mathbf{mc3(int\ i)}}$

$St.3: \ 2.field_1 \leftarrow \{\{\},\{o_1,o1_1\},\{obj.sideEffect(obj)\},\{1\}\}$

$St.3: \ 3.field_1 \leftarrow \{\{\},\{o_1\},\{obj.obj(int),obj2.sideEffect(obj)\},\{4\}\}$

$St.3: \ 4.value_1 \leftarrow \{\{600\},\{o_1\},\{obj.obj(int),obj2.sideEffect(obj)\},\{\}\}$

$St.3: \ \_result_1 \leftarrow \{\{\},\{o_1,2.value_1,3.value_1\},\{obj.sideEffect(obj),$
$obj2.sideEffect(obj)\},\{\}\}$

The difference to the previous example is that here we no longer have to model one method on two different receivers, i.e., locations, but actually two different methods. This leads to the additional FD introduced by method *sideEffect(obj o)* of class *obj2* with location 4.

Example 9.2.2 *Let f1 and f2 be two static fields, which reference objects of type obj and obj2, respectively. The inserted code fragment reads as follows:*

static *obj f1 =* ***new*** *obj(300);*
static *obj2 f2 =* ***new*** *obj2(400);*

Consider now the two methods, mc4() and method mc5(), which are shown in Figures 9.7 and 9.8, respectively. Both call method sideEffect(obj o) on one of the two static fields, whose type is not known inside the modeled method.

```
     int mc4() {
1.     obj o1 = new obj(1000);
2.     return f1.sideEffect(o1);
     }
```

Figure 9.7: Example method *mc4()*

```
     int mc5() {
1.     obj o1 = new obj(1000);
2.     return f2.sideEffect(o1);
     }
```

Figure 9.8: Example method *mc5()*

***ETFDM*:** When computing the *ETFDM* of method *mc4()*, the object referenced by the static field *f1* is not known inside the modeled method. We therefore have to assign it a default location, i.e., location 2. Modeling method *mc4()* the default location 2 is of the static type *obj*. At run-time we also find *f1* to be of type *obj* and thus only have to model method *sideEffect(obj o)* of class *obj*.

$\mathbf{ETFDM}^{()}_{\mathbf{mc4()}}$

$St.2: \ 2.field_1 \leftarrow \{\{\}, \{o1_1, f1_0\}, \{obj.sideEffect(obj)\}, \{1\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\}, \{2.value_0, f1_0\}, \{obj.sideEffect(obj)\}, \{\}\}$

When modeling method *mc(5)* we get a run-time type of *f2* of class *obj2*. Therefore, we model method *sideEffect(obj o)* of class *obj2*. The following FDs contain an additional FD for the new location, i.e., location 3, created in method *sideEffect(obj o)* of class *obj2*.

$\mathbf{ETFDM}^{()}_{\mathbf{mc5()}}$

$St.2: \ 2.field_1 \leftarrow \{\{\}, \{f2_0\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{3\}\}$

$St.2: \ 3.value_1 \leftarrow \{\{600\}, \{f2_0\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\}, \{2.value_0, f2_0\}, \{obj2.sideEffect(obj)\}, \{\}\}$

***DFDM*:** When computing the *DFDM* of method *mc4()* we only know the compile-time type of field *f1*, i.e., *obj*. Because of the fact that at run-time we could as well assign objects of type *obj2* to it, we have to consider both cases. Unlike explicitly created locations the run-time type of default locations can differ from its compile-time type. We therefore have to compute two models, one for method *sideEffect(obj o)* of class *obj* and one for *sideEffect(obj o)* of class *obj2*. The resulting model reads as follows:

$\mathbf{DFDM_{mc4()}}$

$St.2: \ 2.field_1 \leftarrow \{\{\}, \{o1_1, f1_0\}, \{obj.obj(int), obj.sideEffect(obj), obj2.sideEffect(obj)\}, \{1,3\}\}$

$St.2: \ 3.value_1 \leftarrow \{\{600\}, \{f1_0\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\}, \{2.value_0, f1_0\}, \{obj.sideEffect(obj), obj2.sideEffect(obj)\}, \{\}\}$

When modeling method *mc5()* the default location for field *f2* gets a compile-time type of class *obj2*. This type does not have any sub-classes, so no polymorphic method calls are possible. We only have to model method *sideEffect(obj o)* of class *obj2* resulting in the following FDs:

$\mathbf{DFDM_{mc5()}}$

$St.2: \ 2.field_1 \leftarrow \{\{\}, \{f2_0\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{3\}\}$

$St.2: \ 3.value_1 \leftarrow \{\{600\}, \{f2_0\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\}, \{2.value_0, f2_0\}, \{obj2.sideEffect(obj)\}, \{\}\}$

So far we saw that when computing the *ETFDM* of a certain method call, all necessary information can be extracted from the evaluation trace. Neither multiple objects assigned to one reference nor polymorphic method calls pose an ambiguity as far as the declaration of the called method is concerned. The result of that is that in all cases only one model has to be created, i.e., the model of the method being called at run-time.

This, however, is not true for the *DFDM*. Both, the receiver and the declaration of the method being called at run-time cannot always be determined. This leads to multiple models being created during the modeling process, the result of which are more complex FDs in the model of the calling method. One last example, which is designed to combine both ambiguities should illustrate the properties of *ETFDMs* and *DFDMs*, if a given method call is modeled.

Example 9.2.3 *Consider method mc6(int i), which is depicted in Figure 9.9. It combines the problems demonstrated using methods mc3(int i), mc4(), and mc5().*

***ETFDM*:** When computing the *ETFDM* of method *mc6(int i)*, we have to consider two cases, i.e., case 1 ($i > 0$), e.g., 1, and case 2 ($i \leq 0$), e.g., -1. In both cases the exact run-time type of the scope of the method call in statement line 3 (i.e., *o*) can be resolved non-ambiguously. The same is true for the method being called at run-time. In both cases only one method has to be modeled, namely method *sideEffect(obj o)* of class *obj* in case 1 and *sideEffect(obj o)* of class *obj2* in case 2. The resulting FDs are as follows:

```
      int mc6(int i) {
        obj o;
1.      obj o1 = new obj(1000);
2.      if (i > 0) {
2.1.      o = f1; }
        else {
2.2.      o = f2; }
3.      return o.sideEffect(o1);
      }
```

Figure 9.9: Example method *mc6(int i)*

$\mathbf{ETFDM}^{(1)}_{\mathbf{mc6(int\ i)}}$

$St.3: \ 2.field_1 \leftarrow \{\{\}, \{o1_1, o_1\}, \{obj.sideEffect(obj)\}, \{1\}\}$

$St.3: \ \_result_1 \leftarrow \{\{\}, \{o_1, 2.value_0\}, \{obj.sideEffect(obj)\}, \{\}\}$

$\mathbf{ETFDM}^{(-1)}_{\mathbf{mc6(int\ i)}}$

$St.3: \ 2.field_1 \leftarrow \{\{\}, \{o_1\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{3\}\}$

$St.3: \ 3.value_1 \leftarrow \{\{600\}, \{o_1\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{\}\}$

$St.3: \ \_result_1 \leftarrow \{\{\}, \{o_1, 2.value_0\}, \{obj2.sideEffect(obj)\}, \{\}\}$

***DFDM*:** The interesting thing is what happens during the creation of the *DFDM* of method *mc6(int i)*. Now the receiver of the method call in statement line 3 (i.e., *o*) can possibly reference either the object referenced by variable *f1* or the object referenced by variable *f2*. Both these objects are modeled by a default location, default location 2 of type *obj* and default location 3 of type *obj2*. If we chose default location 2 as the receiver of the method call, we find a polymorphic method call. The called method can either be method *sideEffect(obj o)* of class *obj* or method *sideEffect(obj o)* of class *obj2*. On the other hand, if we model the method call with default location 3 as its receiver, only *sideEffect(obj o)* of class *obj2* can be called at run-time. We therefore have to compute 3 different models for the method call in statement line 3. To be more precise we get:

1. *DFDM* of method *obj.sideEffect(obj o)* on default location 2 (var *f1*)
2. *DFDM* of method *obj2.sideEffect(obj o)* on default location 2 (var *f1*)
3. *DFDM* of method *obj2.sideEffect(obj o)* on default location 3 (var *f2*)

The complete *DFDM* of the method call in statement 3 is a combination of all three individual models. It comprises 5 FDs (compared to 2 and 3 FDs in the *ETFDM*). Note that the combination of multiple FDMs is described in Section 6.7.

$\mathbf{DFDM_{mc6(int\ i)}}$

$St.3: 2.field_1 \leftarrow \{\{\}, \{o1_1, o_1\}, \{obj.obj(int), obj.sideEffect(obj), obj2.sideEffect(obj)\}, \{1, 4\}\}$

$St.3: 3.field_1 \leftarrow \{\{\}, \{o_1\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{5\}\}$

$St.3: 4.value_1 \leftarrow \{\{600\}, \{o_1\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{\}\}$

$St.3: 5.value_1 \leftarrow \{\{600\}, \{o_1\}, \{obj.obj(int), obj2.sideEffect(obj)\}, \{\}\}$

$St.3: \_result_1 \leftarrow \{\{\}, \{o_1, 2.value_0, 3.value_0\}, \{obj.sideEffect(obj), obj2.sideEffect(obj)\}, \{\}\}$

Note that now method *sideEffect(obj o)* of class *obj2* is modeled twice, once on default location 2 and once on default location 3. Therefore, 2 new locations (location 4 and location 5) are created as side-effects of this method. This seems to be superfluous, but has to be done, because both new locations are created on different receivers, which could influence the content of the new objects.

Model comparison: What we said in the context of scope resolution (see Section 9.2.1) is also true for determining the declaration of the called method. Whereas with the *ETFDM* only one method has to be modeled on exactly one receiver, multiple methods have possibly to be considered during the computation of a *DFDM*. This, again, results in fewer and smaller FDs in the model of the calling method in the case of the *ETFDM*.

9.2.3 Getting the FDM of the called method

After a method or constructor declaration has been selected for modeling, its FDM is needed. More precisely, what we need is the summarized FD method model, which represents an external view of the called method. Note that in case of the *ETFDM* the method called at run-time can always be determined non-ambiguously. Therefore, we need the summarized FDM of only this method. Generally, this is not true in case of the *DFDM*. If we encounter multiple methods, which are possibly called at run-time, the summarized FDMs of all these methods are needed.

***ETFDM*:** In the case of the *ETFDM*, every time we encounter a method call the called method has to be modeled on demand, regardless whether it has already been modeled or not. The reason for that is that each method call corresponds to its own evaluation sub-trace and different traces might result in different models. Assume a call to a method, which contains a selection statement, whose condition depends on an argument of the method call. Clearly, in this case the resulting model depends on the evaluation trace of the calling method. One big advantage of this is that recursive methods do not have to be dealt with separately. If we assume that the method to be modeled halts on its inputs, we get a finite evaluation trace, which eventually leads to the termination of the modeling algorithm.

***DFDM*:** In the case of the *DFDM* we compute purely static method models, which do not depend on concrete method inputs. This means that we only have to compute a method's model once. We can then store it and use it again, when its FDM is needed. When we encounter a call to a specific method, either its model has already been computed and stored or we have to suspend the current modeling process, compute the model of the called method, and resume the modeling of the calling method. This approach does not work with recursive methods, because of the possibility of an infinite method call sequence during the modeling process. Note that in this section we assume that there are no recursive method calls, neither direct nor indirect ones. This assures that we do not encounter cycles in the chain of method modelings. Chapter 13 is dedicated to the more sophisticated problems arising during the computation of *DFDMs* of recursive methods.

Model comparison: One big advantage of the *ETFDM* over the *DFDM* is that by modeling each method call separately, we get more detailed models of the called method, which can then be transformed to the calling method. As a result this leads to both, fewer and smaller FDs. Another advantage is that recursive methods can be handled without applying additional techniques. These techniques, which are needed in order to handle recursive methods with the *DFDM*, are much more complicated and have higher computational requirements. On the other hand, *DFDMs* are much faster to compute. This should be obvious, if we remember that each method has to be modeled only once, no matter how often it is called by other methods. Furthermore, *DFDMs* can be computed for whole `Java` host environments, whereas *ETFDMs* are only feasible for one particular call of the method to be modeled. In the latter case all models have to be seen only in the context of one evaluation trace and generally cannot be reused for the modeling of other method calls.

9.2.4 Transforming a FDM

We now have to transform the external models of all possibly called methods on all possible receivers to their representations in the calling method. This can be done by successively transforming all method model/receiver pairs and finally combining all transformed models to one model as it will be part of the FDM of the calling method.

Transforming a FDM: If we want to transform a single FD method model, i.e., FDM_m, on a given receiver r, we first compute the summarized FDM of the called method, i.e., $sum(FDM_m)$. We then consider only these EFDs from $sum(FDM_m)$, which are defined for variables, which are visible in the calling method. These EFDs have to be transformed one after another.

Example 9.2.4 *Modeling a call to method sideEffect(obj o) of class obj2 (see Figure 9.3) on a given location l, we compute the summarized FDM of method sideEffect(obj o). If we use a DFDM, the summarized model reads as follows:*

$\mathbf{sum}(\mathbf{FDM}_{sideEffect(obj\ o)})$

$0.field \leftarrow \{\{\}, \{\}, \{obj(int)\}, \{1\}\}$

$1.value \leftarrow \{\{600\}, \{\}, \{obj(int)\}, \{\}\}$

$\_result \leftarrow \{\{\}, \{0.value\}, \{\}, \{\}\}$

Location mapping: Before we explain the exact algorithm of transforming EFDs, we first have a short look at the locations of FDM_m and how they are represented in the calling method's FDM, i.e., FDM_n. In the course of modeling a method call, the following two types of locations are created:

Imported new locations arise in the model of n, if a new location is created in m or imported into m via another method call. In other words, if FDM_m contains an explicitly created or an imported new location l, which is visible in n, an imported new location l' is created. Location l' is of the same type as l, but is assigned a different index due to the consecutive location numbering inside individual methods. Both locations theoretically represent the same object location. Therefore, we define $map(l) = l'$.

Imported default locations arise in the model of m, if a default location is created in m or imported into m via another method call. However, if FDM_m contains a default or an imported default location, l, there is not automatically a new default location created in n. This is only done, if no default location for the variable for which l was created exists. Note that default locations are only used for instance variables, static fields, and parameters. In both cases, whether an imported default location has to be created or not, we write $l' = map(l)$.

Example 9.2.5 *Let us come back to the above example of a call to method sideEffect(obj o) of class obj2 on location l. Internally, two locations are created for method sideEffect(obj o): (1) A default location 0 for the receiver (keyword **this**) of type obj2 and (2) a new location 1 for the class instance creation expression in line 1 of sideEffect(obj o) of type obj. Externally, i.e., in the calling method, location 0 is represented by the location of the method call's scope, i.e., location l. The new location 1 is imported into the calling method and thus becomes an imported new location with a new index n. The new index depends on the current state of the location index counter in the calling method. The following two relations hold:* $map(0) = l$ *and* $map(1) = n$.

Variable mapping: Building on the location mapping, we also define a mapping of variables from their representation in m to their representation in n. This has to be done, because variables can contain locations in their scopes, i.e., in case of instance fields.

Definition 9.2.2 *Let v be a variable defined in method m. We define* $map(v) = l'.x$ *with* $l' = map(l)$ *if v is of the form* $l.x$ *and* $map(v) = v$ *otherwise.*

Example 9.2.6 *If we come back to our running example, we find variable 1.value in the summarized DFDM of method sideEffect(obj o). From the above definition it follows that* $map(1.value) = map(1).value = n.value$.

Imported location keys: Similar to new and default locations, we also define unique keys for imported locations. In contrast to the keys defined in Chapter 6, imported location keys also contain information about the method call trace, through which these locations are imported into a method. More formally, we give the following definitions:

Definition 9.2.3 *Let* MCK *be the set of all method call keys possibly created at run-time. Then a method call path is defined as a (ordered) sequence of method call keys, i.e.,* $\langle mck_1, ..., mck_n \rangle \mid mck_1, ..., mck_n \in MCK$.

Definition 9.2.4 *The key of an imported default location consists of* $\langle v, path \rangle$, *where path is the method call path modeling the full sequence of method calls, through which a default location is imported into a certain method. Similar to default location keys, v stands for the variable for which the default location is created.*

Definition 9.2.5 *The key of an imported new location consists of* $\langle pos, path \rangle$, *where path is the method call path modeling the full sequence of method calls, through which a new location is imported into a certain method. Similar to new location keys pos stands for the source code position, where the new class instance is created.*

Example 9.2.7 *In our running example location 1 of method sideEffect(obj o) is imported into a calling method. There, an imported new location n is created, i.e.,* $map(1) = n$. *Whereas the key of location 1 includes only the source code position of the class instance creation expression in method sideEffect(obj o), say* pos_1, *the key of the imported location n also includes the method call path of the whole importation trace. It reads* $\langle pos_1, path_1 \rangle$, *where* $path1 = \{\langle mc, sideEffect(obj\ o) \rangle\}$ *and mc is the call to method sideEffect(obj o).*

Note that the concepts of method call paths and imported location keys is vital to the handling of recursions, which is described in Chapter 13.

Transforming a single EFD: An EFD, $efd = \langle v, DEP \rangle$, which is visible in the calling method, can now be transformed to its representation in the FDM of the calling method. This is done by creating a new side-effect EFD, $efd' = \langle v', DEP' \rangle$, with the following constituents:

- If v is an instance field and thus has the format $l.x$, we first have to map location l to its representation in the calling method, i.e., $map(l)$. In all other cases $v' = v$. Formally, we write $v' = map(v)$.
- The set of constants or run-time values can be copied to the new EFD unchanged, because constants never change their forms. Therefore, $R' = R$ and $C' = C$.

- When transforming variables on the right-hand side of the EFD, we have to take locations into consideration. Therefore $V' = \{map(v) \mid v \in V\}$.
- In order to model dependencies on whole method declarations we add the called method m to M. We get $M' = M \cup m$.
- Finally, all objects or locations have to be set to their external indices, as they appear in the model of the calling method. Formally, we state: $O' = \{map(o) \mid o \in O\}$ and $L' = \{map(l) \mid l \in L\}$.

Example 9.2.8 *In our example of a call to method sideEffect(obj o) of class obj2 we have to transform all EFDs in the summarized DFDM of method sideEffect(obj o), which reads as follows:*

$\mathbf{sum}(\mathbf{FDM}_{sideEffect(obj\ o)})$

$0.field \leftarrow \{\{\}, \{\}, \{obj(int)\}, \{1\}\}$

$1.value \leftarrow \{\{600\}, \{\}, \{obj(int)\}, \{\}\}$

$\_result \leftarrow \{\{\}, \{0.value\}, \{\}, \{\}\}$

Using the location mappings $map(0) = l$ and $map(1) = n$ we get three new EFDs in the calling method for the method call mc, which read as follows:

$\mathbf{sum}(\mathbf{FDM}_{mc})$

$l.field \leftarrow \{\{\}, \{\}, \{obj(int), sideEffect(obj)\}, \{n\}\}$

$n.value \leftarrow \{\{600\}, \{\}, \{obj(int), sideEffect(obj)\}, \{\}\}$

$\_result \leftarrow \{\{\}, \{l.value\}, \{sideEffect(obj)\}, \{\}\}$

9.2.5 Combining multiple external models

In case of the *DFDM* we possibly deal with multiple receivers and methods being called at run-time. Therefore, we also have to handle multiple external FDMs of the called method, which are transformed to their representations in the calling method. These representations have to be combined to a single (summarized) FDM, which can then be incorporated into the FDM of the calling method. This is done by collecting all side-effect EFDs for a particular variable v and combining their DEP structures by successively applying the union operator. More formally, we write:

Definition 9.2.6 *Let $efd_1 = \langle v, DEP_1 \rangle$ and $efd_2 = \langle v, DEP_2 \rangle$ be two EFDs created for the same variable v, which stem from transforming two different summarized FDM of a called method. Then $comb(efd_1, efd_2) = \langle v, comb(DEP_1, DEP_2) \rangle$ with $comb(DEP_1, DEP_2)$ being the union of all constituents of DEP_1 and DEP_2.*

Note that if the called method m has a return type other than **void**, we also create a EFD for the return value. In case of multiple models, we have to combine all return DEPs in order to get the complete DEP structure of the return variable.

> ***JADE:*** *The* ***JADE*** *system stores the side-effect EFDs of all models to be converted in an intermediate data structure, where they are accumulated until all models are built. After that all side-effect EFDs are incorporated into the current FD method model.*

9.2.6 Adding the method call model

The last step is to incorporate the external FDM of the method call into the FDM of the calling model. This is done, once all summarized models are transformed to their representations in the calling method and all these representations are combined to a single external FDM. If we incorporate an external FDM into the FDM of the calling method, we transform each EFD into a FD. This is done by assigning indices to all variables in the EFD and thus converting these variables to variable occurrences. The concrete indices depend on the current states of the variable index counters in the calling method. The resulting FDs can directly be added to the FDM of the calling method.

Example 9.2.9 *Assume we are modeling a method call in statement 3 of method mc7(int i, int j), which produces the EFD $x \leftarrow y$. We incorporate this EFD into the current DFDM of mc7(int i, int j), which reads as follows:*

$\mathbf{DFDM}_{mc7(int\ i,\ int\ j)}$

$St.1: \ x_1 \leftarrow \{\{\}, \{i_0\}, \{\}, \{\}\}$

$St.2: \ y_1 \leftarrow \{\{\}, \{j_0\}, \{\}, \{\}\}$

We then transform the EFD to a new FD and add it to the DFDM of method mc7(int i, int j). The new FD reads as follows:

$St.3: \ x_2 \leftarrow \{\{\}, \{y_1\}, \{\}, \{\}\}$

9.2.7 Comparing the *ETFDM* and *DFDM*

Finally, we summarize the main differences between *ETFDMs* and *DFDMs* of method and constructor calls and highlight advantages and drawbacks of both model types.

- When computing the *ETFDM* of a given method call the called method has to be modeled on the evaluation trace defined by the calling method. This means that for multiple calls to the same method multiple models

have to be computed. Clearly, this is a time consuming task. Furthermore, if all models of the called methods should be kept (e.g., for hierarchical debugging to step into a called method) this approach requires a lot of memory.

- In the context of *DFDMs* each method has to be modeled only once. It can be stored and retrieved every time a call to this method appears. Therefore, *DFDMs* are better reusable and compute less models of called methods.

- Because of the fact that *ETFDMs* make use of evaluation trace information, the scope of the method call and the declaration of the called method can always be determined non-ambiguously. This means that only one FDM of the called method has to be transformed into its external representation in the calling method.

- When computing the *DFDM* of a method call, multiple models of the called method have to be considered. This is due to potential dependencies, which are part of the model, and polymorphic method calls, which cannot be resolved at compile-time. In a worst case scenario all possibly called methods, i.e., the method declarations of all sub-classes of the static type of the receiver, have to be transformed in combination with all their possible receivers. Finally, the FDs resulting from all modeled methods have to be combined. In this respect it does not only take longer to compute the model of a method call, but also results in more and larger FDs in the model of the calling method.

- In both model types we get sound and complete models of the method call provided that the used FDMs of the called methods are sound and complete. This is, because all FDs are simply imported into the model of the calling method.

- The resulting model of the calling method is minimal in respect to an evaluation trace when computing the *ETFDM*. This is not the case with the *DFDM*, because all run-time scenarios (again, different receivers, polymorphic calls, etc...) have to be considered at compile-time.

9.3 Conditional expressions

Conditional expressions of the form $expr_1$? $expr_2$: $expr_3$ have to be considered separately at expression level, because they introduce control dependencies and potential influences. Logically, they very much resemble selection statements. Therefore, the modeling of these expression will be explained in Section 10.3 together with **if** and **switch** statements.

> ***JADE:*** *The* ***JADE*** *system handles conditional expression of the above format exactly like* ***if*** *statements. This includes a hierarchical modeling process as it is described in Section 10.3.*

Chapter 10

Modeling Statements

So far we have only discussed the modeling of assignments and method calls in detail. This section deals with the modeling of statements, which contain sub-blocks. These statements have to be treated differently, because all sub-blocks are modeled separately and then combined to the statement model of the statement in question. According to the `Java` Language Specification [16] the following statements contain sub-blocks and are therefore discussed in this section:

1. Blocks
2. Selection statements (**if** and **switch** statements)
3. Loop statements (**do**, **for**, and **while** statements)
4. Synchronization statements (keyword **synchronize**)
5. Try statements (keyword **try**)

Note that all other statements can be modeled by an empty FDM (e.g., **return**) or the expression model of a single sub-expression (e.g., **return** expr). This topic is shortly discussed in Section 6.5.

10.1 Modeling principles

Generally, statements containing sub-blocks are modeled in two steps. First, all sub-blocks are modeled separately. Second, all sub-block models together with expressions contained in the statement, if any, are combined to create the top-level statement model of the statement in question. This approach leads to the creation of hierarchical models, where each level in the model hierarchy can be associated with exactly one block of the modeled source code.

Creating sub-block models: In order to create a FDM of a statement's sub-block, we define an auxiliary method declaration, whose body is equivalent to the sub-block to be modeled. We can then directly use the FDM of the auxiliary method as the sub-block's FDM at the next level in our model hierarchy.

However, the full sub-block model is not sufficient. Moreover, we need the summary of the sub-block model, i.e., the summarized FDM of the auxiliary method, in order to get all FDs, which are visible at the level of the modeled statement. The computation of summarized models is described in Section 6.6.

Combining multiple sub-block models: Once we have all sub-block models, we have to compute the top-level statement model of the original statement. This is done in two ways:

1. In some cases it is possible to directly use the model of a sub-block as the top-level statement model, e.g., when modeling nested blocks. Note that in this case the resulting model is not hierarchical, because the model structure no longer strictly reflects the source code's block hierarchy.

2. In most cases the sub-block models have to be combined to a single statement model at top-level. Note that in this case all sub-block models have to be stored separately, because they constitute the next level of our model hierarchy.

In the following sections we show how to model various statements. It is assumed that all sub-block models are present in their full and summarized versions. Therefore, the main issues are to combine these models to the top-level statement model of the considered statement.

10.2 Blocks

Nested sub-blocks are the easiest source code structures to handle, because they can be modeled in a hierarchical and a direct way. If we chose to model blocks in a hierarchical way, the full sub-block model has to be added to the model hierarchy. Further on, the summarized sub-block model can be used as the top-level statement model of the modeled block.

The other option is to directly incorporate the full sub-block model into the top-level model by using the full sub-block model as statement model of the block. This can be done, because nested blocks only result in additional data dependencies, but no control dependencies. The result of the latter technique is a flatter hierarchy, which as mentioned before no longer totally reflects the original source code block structure. Since there arise no dependencies on a block's top-level, i.e., if the block is regarded as a statement, the computation of the *ETFDM* works exactly like the computation of the *DFDM*.

As we will see in Part III of this work, a flatter hierarchy has to be preferred, when using the FDM for debugging, because more diagnoses can be excluded at the top-level and less overhead occurs when shifting from one level in the hierarchy to another.

JADE: The JADE system currently models blocks in a non-hierarchical way, because of the better applicability of the resulting models to the debugging process.

10.3 Selection statements

Generally speaking, selection statements are statements, which have more than one sub-block plus a condition. At run-time the evaluation value of the condition tells the run-time environment, which of the sub-blocks should be evaluated. Selection statements are **if** and **switch** statements. The conditional expression of the form $expr_1 \, ? \, expr_2 \, : \, expr_3$ is no statement, but it can be modeled exactly like **if** statements. This is why it is discussed in this section, too.

*JADE: the JADE system is able to model **if** statements and conditional expressions. **Switch** statements are not yet implemented, but, as we will see later, could follow the same principles as **if** statements.*

10.3.1 Computing FDMs of selection statements

***ETFDM*:** When we want to compute the *ETFDM* of a selection statement, a concrete evaluation trace has to be known. We therefore assume that the evaluation value of the statement's condition is also known, which tells us, which sub-block, i.e., branch, of the selection statement is evaluated at run-time. It is only this branch we have to model. The exact modeling process works as follows:

1. Compute all FDs arising from the selection statement's condition. These FDs can be side-effect FDs, which are imported into the current model via method calls inside the condition.

2. Compute the model of the evaluated branch. Since we only consider one branch, we can treat this block just like a nested block (see Section 10.2) and model it hierarchically or non-hierarchically.

3. Finally, all constants, variables, and method calls in the condition have to be added to all FDs arising from the evaluated branch. This has to be done in order to model the control dependencies of the statement.

Consider, for example, method *if1(int i)*, which is depicted in Figure 10.1. If *if1(int i)* gets called with a value of $i > 0$, e.g., 1, then statement 1.1 is evaluated and an object of type *obj* with a value of 100 is created. The return value of *if1(int i)* is 100. The complete *ETFDM* of *if1(int i)* for $i > 0$ reads as follows:

```
      int if1(int i) {
        obj o;
1.      if (i > 0) {
1.1.      o = new obj(100);
        }
        else {
1.2.      o = new obj(200);
        }
2.      return o.value;
      }
```

Figure 10.1: Example method *if1(int i)*

$\mathbf{ETFDM}^{(\mathbf{1})}_{\mathbf{if1(int\ i)}}$

$St.1: \ 1.value_1 \leftarrow \{\{0,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ o_1 \leftarrow \{\{0\},\{i_0\},\{obj.obj(int)\},\{1\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\},\{1.value_1, o_1\},\{\},\{\}\}$

Now assume *if1(int i)* gets called with a value of $i \leq 0$, e.g., -1. In this case statement 1.2 gets evaluated. Again, an object of type *obj* is created, but this time with a value of 200. Therefore, the return value of *if1(int i)* is 200. Now the complete *ETFDM* of *if1(int i)* is somewhat different:

$\mathbf{ETFDM}^{(\mathbf{-1})}_{\mathbf{if1(int\ i)}}$

$St.1: \ 1.value_1 \leftarrow \{\{0,200\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ o_1 \leftarrow \{\{0\},\{i_0\},\{obj.obj(int)\},\{1\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\},\{1.value_1, o_1\},\{\},\{\}\}$

Finally, we may want to compute the combined *ETFDM*, i.e., $ETFDM^{(1),(-1)}_{\text{if1(int i)}}$. As explained in Chapter 7, this model covers all FDs arising from variable occurrences, method declarations, and objects, but does not include all run-time values. Therefore, the following model can be seen as a good approximation of $CFDM_{\text{if1(int i)}}$.

$\mathbf{ETFDM}^{(\mathbf{-1,1})}_{\mathbf{if1(int\ i)}}$

$St.1: \ 1.value_1 \leftarrow \{\{0,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ 2.value_1 \leftarrow \{\{0,200\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ o_1 \leftarrow \{\{0\},\{i_0\},\{obj.obj(int)\},\{1,2\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\},\{1.value_1, 2.value_1, o_1\},\{\},\{\}\}$

Note that in each of the two individual models a single location indexed one is created. These locations, however, are different locations. When computing the combined FDM at least one of the two has to be renamed to retain the full dependency structure.

***DFDM*:** To compute the *DFDM*, not only data and control dependencies have to be considered, but also potential influences (see Section 6.3). This has to be done, because we do not know the run-time value of the condition and therefore have to take all branches of a selection statement into consideration. In other words, we now compute all FDs, which might arise from either of the statement's branches, not just from the one evaluated at run-time. It is easy to see that all sub-blocks have to be modeled and summarized. Further on, we can no longer build a flat model, but have to create a hierarchical one. The key question remains, how multiple sub-block models can be combined into a top-level statement model. Algorithm 10.3.1 answers that question:

Algorithm 10.3.1

- Compute the full FDMs for all branches of the selection statement.
- Summarize all FD branch models as described in Section 6.6.
- For each variable v, which possibly changes its value in any of the statement's branches, compute a new FD by building the union of the FDs of v in all branches, where such an FD exists. This can be done by combining the summarized FDMs of all branches.
- For each variable v, which does not change its value in all branches, introduce a so called self dependency, i.e., $v_n \leftarrow v_{n-1}$. This has to be done to model the case in which the value of v is not changed during run-time and therefore only depends on the value of v before the selection statement.
- The right-hand sides of all FDs of the combined model have to be merged with the DEP structure arising from the statement's condition. This has to be done to cover all control dependencies, i.e., to model the influences of the condition's evaluation value on the branch executed at run-time.
- Finally, the resulting summarized FDM can be incorporated into the FDM containing the selection statement. This is done by assigning new indices to all variables in the summarized model.

If we apply Algorithm 10.3.1 to our example program we get the following FDM for method *if1(int i)*. In contrast to the *ETFDMs* computed for a single branch two locations is created. Because of the fact that it is unknown at compile-time, which of the two locations are created by the `Java` system at run-time, the return value of *if1(int i)* depends on both locations.

$\mathbf{DFDM_{if1(int\ i)}}$

$St.1: \ 1.value_1 \leftarrow \{\{0,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ 2.value_1 \leftarrow \{\{0,200\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.1: \ o_1 \leftarrow \{\{0\},\{i_0\},\{obj.obj(int)\},\{1,2\}\}$

$St.2: \ \_result_1 \leftarrow \{\{\},\{1.value_1, 2.value_1, o_1\},\{\},\{\}\}$

Note that the *DFDM* and the combined *ETFDM* of method *if1(int i)* are equivalent, except for the run-time values in the *ETFDM*. This is obvious, if we remember that in both cases all possible run-time scenarios are covered by the resulting FDM.

> ***JADE:*** *When computing the FDM of an* ***if*** *statement, the* ***JADE*** *system performs hierarchical modeling, no matter whether the ETFDM or DFDM is created. In the case of the ETFDM, the evaluated sub-block could also be modeled in a direct way. However, this is not done, because it makes both models better comparable and leaves the implementation more uniform.*

10.3.2 Introducing self dependencies

Cases, where a variable v is assigned a value in some but not all of a selection statement's branches, have to be dealt with separately. In this case we introduce self dependencies of the form $v_n \leftarrow v_{n-1}$ in order to model a run-time scenario, where a branch not changing v is executed and thus the value of v after the execution of the selection statement is equal to the value of v before the statement's execution.

Consider the `Java` method *if2(int i)*, which is depicted in Figure 10.2. In the selection statement in line 4 two variables, a and b, are used on the left-hand side of an assignment. Whereas in the then-branch the values of both variables are changed, the else-block only affects the value of a leaving b unaltered.

***ETFDM*:** The creation of the *ETFDM* is straightforward and easy to understand. In case 1 ($i > 0$, e.g., $i = 1$) we get two FDs arising from the then-block in lines 4.1 and 4.2, whereas the modeling of the else-block in line 4.3 (case 2: $i \leq 0$, e.g., $i = -1$) only produces one FD for variable a. The two models read as follows:

$\mathbf{ETFDM^{(1)}_{if2(int\ i)}}$

$St.4: \ a_2 \leftarrow \{\{0\},\{b_1, i_0\},\{\},\{\}\}$

$St.4: \ b_2 \leftarrow \{\{0\},\{c_1, i_0\},\{\},\{\}\}$

```
      int if2(int i) {
1.      int a = 1;
2.      int b = 2;
3.      int c = 3;
4.      if (i > 0) {
4.1.      a = b;
4.2.      b = c; }
        else {
4.3.      a = c; }
5.      return a;
      }
```

Figure 10.2: Example method *if2(int i)*

$\mathbf{ETFDM}^{(-1)}_{\mathbf{if2(int\ i)}}$

$St.4: \; a_2 \leftarrow \{\{0\}, \{c_1, i_0\}, \{\}, \{\}\}$

Note that in the second model (case 2) no FD for variable b is created at all. Therefore, we do not have to create a self dependency, because all variables not covered by any FDs are implicitly assumed to remain unchanged.

***DFDM*:** When now considering the *DFDM* we find that variable b is only used in the selection statement's then-branch and therefore only depends on c. In order to explicitly cover the case, where the else-block is executed and b stays unchanged, we introduce the FD $b_2 \leftarrow b_1$, which leads to the following FDM for the selection statement in line 4:

$\mathbf{DFDM}_{\mathbf{if2(int\ i)}}$

$St.4: \; a_2 \leftarrow \{\{0\}, \{c_1, b_1, i_0\}, \{\}, \{\}\}$

$St.4: \; b_2 \leftarrow \{\{0\}, \{c_1, b_1, i_0\}, \{\}, \{\}\}$

10.3.3 Other selection statements

The JLS [16] defines a second type of selection statement, i.e., the **switch** statement and a selection expression, i.e., the conditional expression of the form $expr_1 \; ? \; expr_2 \; : \; expr_3$. Both source code structures behave very much like the **if** statement described above. Conditional expressions can be modeled exactly like **if** statements, only at expression level. **Switch** statements can be modeled by applying the algorithms explained above to the general case of n branches. Problems might only arise due to a complex structure of **break** statements. In these cases it is not guaranteed that **switch** statements are fully evaluated after the

execution of one of its branches. The modeling of labels and **break** & **continue** statements is not discussed in this work in detail.

> ***JADE:*** *The* ***JADE*** *system currently models conditional expressions exactly like* ***if*** *statements. The only problem at the moment is that the code instrumentation component computing the evaluation traces only works at statement level and therefore cannot determine the run-time value of a conditional expression's condition. This is why the* ***JADE*** *system currently always computes DFDMs for conditional expressions.* ***Switch*** *statements are currently not implemented, because* ***break*** *&* ***continue*** *statements are not yet supported by the* ***JADE*** *modeling component.*

10.3.4 Model comparison

Selection statements uncover one of the main advantages of *ETFDMs* in comparison to *DFDMs*. If we only need a model for one particular evaluation trace, with the *ETFDM* we get a substantially lower amount of FDs arising from the sub-blocks of the selection statement. In the case of **if** statements, for instance, we expect a reduction of the analyzed source code of nearly 50 percent provided that an else-branch exists (and the else-branch contains approximately the same number of statements as the then-branch). This results in a much lower number of FDs compared to a purely static analysis like the *DFDM*. If the else-branch is empty, we do not need self dependencies, which also reduces the complexity of the resulting model. The same is true for conditional expressions, whereas the advantage of *ETFDMs* in this respect should even be greater in the case of **switch** statements containing more than two branches.

Furthermore, *ETFDMs* can be shown to be sound in all cases, which is not true for *DFDMs*. As already mentioned *DFDMs* compute only an approximation of the *CFDM* in cases of assignments to variables, which do not appear in all branches of the selection statement.

A major drawback of *ETFDMs* in comparison to *DFDMs* is that their creation is based on the existence of an evaluation trace. As it is the case with *ETFDMs* of all source code structures, these models cannot be constructed if the appropriate run-time information is missing. Furthermore, the branches of a selection statement have to be modeled for each sub-trace in the case of the *ETFDM*, which means that the creation of *ETFDMs* of selection statements is much slower and needs a lot more memory than the creation of *DFDMs*. Finally, if a whole host environment model, which covers all run-time scenarios, is to be created, the *DFDM* is the only possible solution (see Section 7.6).

10.4 Loop statements

One of the key aspects of every programming language (and thus of any meaningful model) are loops. The `Java` Language Specification [16] provides the following kinds of loop statements:

Do statements (like all loop statements) have a body and a condition. When a **do** loop is encountered by the JVM during run-time, its body and condition are repeatedly executed starting with the loop body. This means that the body of a **do** statement is executed at least once.

For statements have two additional features apart from a body and a condition, i.e., an initializer and an update block. A **for** loop is evaluated starting with its initializer. Then its condition, body, and update block are evaluated in the stated sequence. Note the following peculiarities of **for** loops: (1) The initializer is always executed first. It can be seen as an own block at the same level as the loop statement. Nevertheless, all variables defined in the initializer are only visible inside the **for** statement. The initializer is executed only once. (2) The body of a **for** loop is not necessarily executed. This can be the case if the loop's condition evaluates to `false` right after the initializer is executed, i.e., before the first iteration. (3) The update block of a **for** loop is evaluated after each iteration. Logically, it is part of the body and not at the same level as the **for** loop.

While statements work very much like **do** loops. The only difference is that its execution starts with the evaluation of its condition rather than its body. This means that in contrast to **do** loops the body of a **while** statement is not necessarily executed.

The following sections describe in detail the computation of *ETFDMs* and *DFDMs* for loop statements. At the beginning, general problems underlying the modeling of loops and modeling principles are discussed. We then show how to model **while** loops in detail and present examples highlighting various properties of the resulting models. Finally, we describe how the presented techniques can be used to model **do** and **for** statements.

10.4.1 Modeling principles

In order to demonstrate the key issues in modeling loop statements, we first look at the FD structure of **while** statements. A **while** statement is executed starting with its condition. This means that possibly its body never gets executed at all, if the condition evaluates to `false` at the very beginning. On the other hand its condition might evaluate to `true` i times, which results in the body being executed i times, too. In this case we say that i iterations of the **while** loop are executed. Note that there is also the eventuality that a **while** loop's condition always evaluates to `true` resulting in an infinite loop. This leads to the non-termination of the whole program, contradicting our initial assumptions (see Section 5.2).

Consider, for instance, method *while1(int i)*, which is depicted in Figure 10.3. In particular, we are interested in the FD structure arising from the **while** statement in line 5 for various numbers of iterations being executed ($i \geq 0$). Obviously, all variables used in the loop's body (a, b, c, and i) depend on variable i, because i appears in the loop's condition. But what are the FDs between the variables a, b, c, and d? The following cases have to be considered:

```
      int while1(int i) {
1.      int a = 1;
2.      int b = 2;
3.      int c = 3;
4.      int d = 4;
5.      while (i > 0) {
5.1.      a = b;
5.2.      b = c;
5.3.      c = d;
5.4.      i − −; }
6.      return a+b+c;
      }
```

Figure 10.3: Example method *while1(int i)*

Case 1 ($i = 0$)**:** The body of the **while** loop is never executed. Obviously, no new FDs can arise. We can alternatively say that each variable depends on itself, since its value stays unaltered. This fact is represented by the following FDM. Note that here, for the sake of clarity, only variable dependencies (without indices) are given.

$\mathbf{FDM_{while1(int\ i)}}$

$St.\,5.1.:\ a \leftarrow \{a\}$

$St.\,5.2.:\ b \leftarrow \{b\}$

$St.\,5.3.:\ c \leftarrow \{c\}$

Case 2 ($i = 1$)**:** Here the body of the **while** statement is executed exactly once. The arising FDs can be taken directly from the source code. They read as follows:

$\mathbf{FDM_{while1(int\ i)}}$

$St.\,5.1.:\ a \leftarrow \{b\}$

$St.\,5.2.:\ b \leftarrow \{c\}$

$St.\,5.3.:\ c \leftarrow \{d\}$

Case 3 ($i = 2$)**:** If the body of the **while** loops is executed twice, some of the FDs change. The new FDs are:

$\mathbf{FDM_{while1(int\ i)}}$

$St.5.1.: a \leftarrow \{c\}$

$St.5.2.: b \leftarrow \{d\}$

$St.5.3.: c \leftarrow \{d\}$

Case 4 ($i > 2$): If the loop's body is executed more often than twice we get an FD structure which does not change any more with an increasing number of iterations. This behavior should be quite obvious, if one looks at the fact that now all variables solely depend on d, which is not altered itself in the body of the **while** loop. The FDs for $i > 2$ are as follows:

$\mathbf{FDM_{while1(int\ i)}}$

$St.5.1.: a \leftarrow \{d\}$

$St.5.2.: b \leftarrow \{d\}$

$St.5.3.: c \leftarrow \{d\}$

Whereas in each iteration new FDs arise, other FDs disappear due to the repeated execution of the variable assignments found in the loop body. In each iteration we get three FDs with three variables on their right-hand sides. If we compute the closure of all four FDMs we get a static model comprising all 4 cases. Its FDs are:

$\mathbf{FDM_{while1(int\ i)}}$

$St.5.1.: a \leftarrow \{a, b, c, d\}$

$St.5.2.: b \leftarrow \{b, c, d\}$

$St.5.3.: c \leftarrow \{c, d\}$

Alltogether we find 9 variables on the right-hand sides of the three FDs. Whereas we now manage to incorporate all four cases, i.e., all run-time scenarios as far as variable dependencies are concerned, into one model, the model becomes less precise and its FDs are much larger.

10.4.2 Computing the FDs of a loop

The FDs of a **while** statement change with an increasing number of loop iterations. The remaining question is how these FDs can be computed in the general case for both model types, the *ETFDM* and the *DFDM*. This can be achieved by rewriting the loop as a sequence of nested **if** statements with the following properties:

```
while (i > 0) {
    a = b;
    b = c;
    c = d;
    i − −;
}
```

(a) Loop body

```
if (i > 0) {
    a = b;
    b = c;
    c = d;
    i − −;
    if (i > 0) {
      a = b;
      b = c;
      c = d;
      i − −;
    }
}
```

(b) Nested **if** structure

Figure 10.4: Transforming a loop statement into nested **if** statements

1. The condition of each **if** statement is equivalent to the loop condition.
2. The **if** statements' then-branches are the concatenation of the statements of the loop body and the inner nested **if** statements.
3. The **if** statements' else-branches are empty blocks.

The only free parameter of this approach is the depth of the nested **if** structure. This parameter, we call it *nestsize*, depends on the type of model to be computed. Its computation is described in the following paragraphs.

Example 10.4.1 *Figure 10.4 (a) shows, again, the body of the loop statement of method while1(int i) (see Figure 10.3). The result of the transformation into nested* ***if*** *statements with* $nestsize = 2$ *is depicted in Figure 10.4 (b).*

Note that the transformation of loop statements into a sequence of nested **if** statements does not result in an executable `Java` program, if local variables appear in the loop body. This is, because `Java` does not support the definition of multiple local variables with the same name within the same variable scope. However, this problem can easily be solved by either eliminating all local variable declarations in the nested **if** structure except for the outermost **if** statement or ignoring these declarations during the modeling process.

***ETFDM*:** When we compute the *ETFDM* of a given **while** loop, we once again can rely on the evaluation trace we are building our model on. The trace information directly gives us the number of iterations of the **while** loop executed at run-time. Let now the number of iterations, as taken from the evaluation trace, be i. We can then construct a nested **if** structure with $nestsize = i + 1$, which is completely equivalent to the **while** statement.

Proposition 10.4.1 *When the exact number of loop iterations executed at runtime is known to be i, we construct a nested* ***if*** *structure with $nestsize = i + 1$. This structure is computationally equivalent to the original loop statement for the given evaluation trace.*

Proof:

- If there are i iterations, then the loop condition evaluates to `true` exactly i times. The loop body is executed i times, too.
- Since the conditions of all nested **if** statements are equivalent to the loop condition, a nested structure with $nestsize = i$ also executes the loop condition and its body (which was copied to the **if** statements then-branch) i times in the correct order.
- We then need one extra **if** statement to guarantee the evaluation of the loop condition for iteration $i + 1$. This is especially important if the loop condition contains side-effects, which influence the resulting FD model.
- Since the evaluation of the loop condition in iteration $i + 1$ evaluates to `false`, the else-branch of the innermost **if** statement is executed. Because this branch is empty, it has the same effect as the **while** loop being terminated.

Q.E.D.

Since both structures, the original **while** loop and the nested **if** structure, are computationally equivalent, it is guaranteed that the correct FDM can be computed by applying this technique. This can be demonstrated by getting back to the above example. The *ETFDMs* for various numbers of iterations as computed by the `JADE` modeling components are given below. Note that for $(i = 0)$ the loop body is never executed and no FDs arise from the **while** statement, i.e., $ETFDM^{(0)}_{\text{while1(int i)}}$ is empty.

$\mathbf{ETFDM^{(1)}_{while1(int\ i)}}$

$St.5: \; a_2 \leftarrow \{\{0\}, \{b_1, i_0\}, \{\}, \{\}\}$

$St.5: \; b_2 \leftarrow \{\{0\}, \{c_1, i_0\}, \{\}, \{\}\}$

$St.5: \; c_2 \leftarrow \{\{0\}, \{d_1, i_0\}, \{\}, \{\}\}$

$St.5: \; i_1 \leftarrow \{\{0, 1\}, \{i_0\}, \{\}, \{\}\}$

$\mathbf{ETFDM}^{(2)}_{\mathbf{while1(int\ i)}}$

$St.5: \ a_2 \leftarrow \{\{0,1\},\{c_1,i_0\},\{\},\{\}\}$

$St.5: \ b_2 \leftarrow \{\{0,1\},\{d_1,i_0\},\{\},\{\}\}$

$St.5: \ c_2 \leftarrow \{\{0,1\},\{d_1,i_0\},\{\},\{\}\}$

$St.5: \ i_1 \leftarrow \{\{0,1\},\{i_0\},\{\},\{\}\}$

$\mathbf{ETFDM}^{(3)}_{\mathbf{while1(int\ i)}}$

$St.5: \ a_2 \leftarrow \{\{0,1\},\{d_1,i_0\},\{\},\{\}\}$

$St.5: \ b_2 \leftarrow \{\{0,1\},\{d_1,i_0\},\{\},\{\}\}$

$St.5: \ c_2 \leftarrow \{\{0,1\},\{d_1,i_0\},\{\},\{\}\}$

$St.5: \ i_1 \leftarrow \{\{0,1\},\{i_0\},\{\},\{\}\}$

***DFDM*:** When computing the *DFDM* of a **while** statement, we face the problem that we do not know the exact number of iterations being executed at runtime. This also means that in contrast to the *ETFDM* the parameter *nestsize* is not known a priori, but has to be computed using only static information about the source code. What we see from the above example is that the FDs arising from a **while** loop only change up to a certain number of iterations and stay the same afterwards. Any nesting depth above this number results in a nested **if** structure, which in general is not computationally equivalent to the **while** statement, but produces exactly the same FDs as the loop statement.

As described in Chapter 6, the summarized FDM of a given block b can be depicted as a directed graph (possibly with cycles), where an edge leading from variable v to variable w means that the value of v influences the value of w, i.e., w depends on v. Obviously, all FDs arising from block b can directly be seen from such a digraph. Let us now come back to our running example from Figure 10.3. Figure 10.5 shows the digraph created for the body of the loop at statement line 5. If we are interested in all FDs arising during the first iteration of the loop statement, we start at an arbitrary node, say c, and follow an outgoing edge to another node b. We see that b depends on c in iteration 1.

If, on the other hand, we are interested in all FDs arising after 2 iterations, we, again, start at node c, but this time follow a path of length two to node a via node b. We see that the graph now produces the FD $a \leftarrow c$ after iteration 2. It should now be easy to understand that collecting the FDs arising from all possible paths between any two variables leads to a complete FDM of block b. The problem of determining an optimal value for the parameter *nestsize* now equals the problem of finding the maximum of all shortest paths between the nodes of a given digraph.

We can determine an optimal value for *nestsize* by performing a purely static analysis on the source code. The exact algorithm can be summarized as follows:

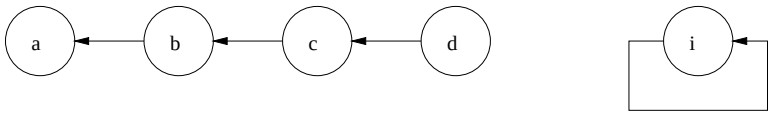


Figure 10.5: Digraph of the **while** body of Figure 10.3

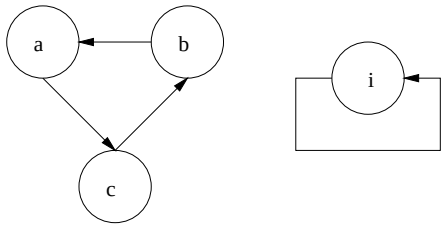


Figure 10.6: Digraph of the **while** body of Figure 10.7

Algorithm 10.4.1

- Compute the complete FDM of the loop body b, i.e., FDM_b
- Compute the summarized FDM of b, i.e., $sum(FDM_b)$
- Construct a directed graph with V containing all variables appearing on either side of an FD in $sum(FDM_b)$ and E containing an edge from v to w iff $w \leftarrow v \in sum(FDM_b)$.
- Compute the length of the shortest path p_{ij} from node i to node j $\forall i, j \in V$. If no path between i and j exists, assume $p_{ij} = 0$.
- Compute *nestsize* as the maximum length of all shortest paths, i.e., $nestsize = max(p_{ij})$ $\forall i, j \in V$.

Proposition 10.4.2 *When the exact number of loop iterations at run-time is not known, i.e, a purely static analysis on the source code is performed, we construct a nested* ***if*** *structure with* $nestsize = max(p_{ij})$ $\forall i, j \in V$. *This structure is computationally not equivalent to the original loop statement, but the sets of FDs arising from both structures are equivalent.*

It should be intuitive that, if we follow all paths of a length of up to *nestsize*, we get all FDs arising from the loop body at the loop's top-level. Using Algorithm 10.4.1 we can further employ existing algorithms taken from graph theory in order to solve the so called all pairs shortest path (APSP) problem. Algorithms that find all shortest paths in $O(n^3)$ time, where n denotes the number of nodes in the digraph, are presented in [1].

```
      int while2(int i) {
1.      int a = 1;
2.      int b = 2;
3.      int c = 3;
4.      int d = 4;
5.      while (i > 0) {
5.1.      a = b;
5.2.      b = c;
5.3.      c = a;
5.4.      i − −; }
6.      return a+b+c;
      }
```

Figure 10.7: Example method *while2(int i)*

Example 10.4.2 *If we look at the digraph in Figure 10.5 we find that nestsize for the modeling of the loop statement of our example method while1(int i) has to be at least 3. This is, because variable d influences variable a through a path of the length 3, i.e., from iteration 3 onwards.*

Example 10.4.3 *Consider another example, method while2(int i), which is a small modification of while1(int i). Its source code is depicted in Figure 10.7. Here in line 5.3 variable c depends on a instead of d, which leads to the summarized body model represented by the digraph shown in Figure 10.6. In contrast to the digraph in Figure 10.5 this graph now contains a cycle, because c directly depends on a and a indirectly depends on c through the two variable assignments in lines 5.1 and 5.2. Despite of the fact that the body of while2(int i) is very similar to the one of while1(int i) Figure 10.6 shows that all variable dependencies are produced in the first two iterations, i.e., nestsize* $= 2$.

If we do not want to use the Algorithm 10.4.1 to compute an optimal value of *nestsize*, another way is to estimate the parameter by a value *nestsize*$'$. It then has to be guaranteed that the new value *nestsize*$'$ is greater than or equal to *nestsize*, i.e., *nestsize*$' \geq$ *nestsize*, because only then all FDs are guaranteed to be included in the resulting model. The following approaches for an estimation of *nestsize* can be used:

- The number of FDs in the *DFDM* of the loop body. In this case *nestsize*$' \geq$ *nestsize*, because after *nestsize*$'$ iterations all variable dependencies are propagated through the whole FDM at the latest. However, if one variable appears on the left-hand side of two or more FDs this approach cannot be optimal, because the summary of all FDs (as used in Algorithm 10.4.1) eliminates these FDs.

- The number of variables or better the number of variables on the left-hand side of an FD. Again, *nestsize*$' \geq$ *nestsize*, because the number of

variables cannot exceed $|V|$. The problem here is that a worst case scenario is assumed, in which all nodes of the graph are placed in a linear order. In our example we would get $nestsize' = 4$, which stems from the assumption that an extra iteration is needed to produce the dependency $i \leftarrow i$, which in fact is produced in iteration 1.

JADE: *The* ***JADE*** *system currently estimates the parameter nestsize by using a function nestsize(b) for a given body of a* ***while*** *loop, i.e., b. Function nestsize(b) counts all top-level assignments and method calls of b. Nested loop statements in b are counted as 1, because these structures are assumed to be transformed into nested* ***if*** *structures themselves during the hierarchical modeling process. Nested* ***if*** *statements are counted as the sum of all functions nestsize(t) computed for all of their branches t. More formally, we write: Let $m(s)$ be the number of side-effects introduced through method calls in a certain statement s. Then $nestsize(b) = \Sigma_{s \in b} nestsize(s)$ with*

- $nestsize(s) = 1 + m(s)$ *if s is a variable assignment or loop statement.*
- $nestsize(s) = \Sigma_{branch \in s} nestsize(branch) + m(s)$ *if s is a selection statement.*
- $nestsize(s) = m(s)$ *in all other cases*

Note that the exact algorithm has been proposed in [28].

If we now use a $nestsize \geq 3$ to convert the loop statement in method *while1(int i)* into nested **if** statements and then model the nested structure as described in Section 10.3, we eventually get the following *DFDM* as a top-level model for the loop in line 5 of Figure 10.3.

$\mathbf{DFDM_{while1(int\ i)}}$

$St.5: \; a_2 \leftarrow \{\{0,1\},\{a_1,b_1,c_1,d_1,i_0\},\{\},\{\}\}$

$St.5: \; b_2 \leftarrow \{\{0,1\},\{b_1,c_1,d_1,i_0\},\{\},\{\}\}$

$St.5: \; c_2 \leftarrow \{\{0,1\},\{c_1,d_1,i_0\},\{\},\{\}\}$

$St.5: \; i_1 \leftarrow \{\{0,1\},\{i_0\},\{\},\{\}\}$

Note that in the above model all FDs are present, especially the FD $a_2 \leftarrow d_1$, which is only created in iteration 3 at run-time and can thus only be reproduced with $nestsize \geq 3$. Note further that not only variable i depends on itself, but also all other variables. These self dependencies are necessary to model the case, in which the loop is never executed, i.e., $i = 0$. Technically, these FDs are produced by the empty else-branches of the nested **if** statements (see Section 10.3).

```
      int while3(int t) {
1.      int a = 1;
2.      int b = 2;
3.      int c = 0;
4.      int i = 0;
5.      while (i < t) {
5.1.      if (i%2 == 0)
5.1.1.      c = a;
          else
5.1.2.      c = b;
5.2.      i++; }
6.      return c;
      }
```

Figure 10.8: Example method *while3(int i)*

10.4.3 Sub-traces for loop bodies

One point that we have not been discussing yet is the fact that at run-time we get different evaluation traces for different loop iterations. When computing the *DFDM* this should not make a difference, but with the *ETFDM* it leads to different FD body models in the various iterations of a loop.

Consider example method *while3(int t)* (see Figure 10.8), where an **if** statement (line 5.1) is nested in a loop statement (line 5). Clearly, now the decision, which branch of the selection statement is executed at run-time, depends on the current loop iteration and its evaluation trace. Here lines 5.1.1 and 5.1.2 are executed alternatively depending on the current value of the loop counter i. Therefore, the FD body models of the loop statement look different for even and odd values of i. The resulting model of the whole **while** loop depends on the total number of iterations performed and thus on the input parameter t.

***ETFDM*:** The following two *ETFDMs* are computed for $t = 1$ and $t = 2$, respectively. Note that in the case $t = 0$ no new FDs arise from the loop statement.

$\mathbf{ETFDM}^{(1)}_{\mathbf{while3(int\ t)}}$

$St.5: \; i_2 \leftarrow \{\{1\}, \{i_1, t_0\}, \{\}, \{\}\}$

$St.5: \; c_2 \leftarrow \{\{0, 2\}, \{a_1, i_1, t_0\}, \{\}, \{\}\}$

$\mathbf{ETFDM}^{(2)}_{\mathbf{while3(int\ t)}}$

$St.5: \; i_2 \leftarrow \{\{1\}, \{i_1, t_0\}, \{\}, \{\}\}$

$St.5: \; c_2 \leftarrow \{\{0, 1, 2\}, \{b_1, i_1, t_0\}, \{\}, \{\}\}$

```
      list while4(int i) {
1.      list l = list.create(i);
2.      while (i > 0) {
2.1.      i − −;
2.2.      obj o = new obj(100);
2.3.      l.set(i,o);
        }
3.      return l;
      }
```

Figure 10.9: Example method *while4(int i)*

***DFDM*:** When computing the *DFDM* all FDs are generated. Variable c now depends on i and t (i.e., the loop condition), a (modeling odd numbers of iterations), b (modeling even numbers of iterations), and c (modeling the special case $t = 0$). The resulting model reads as follows:

$\mathbf{DFDM_{while3(int\ t)}}$

$St.5: \; i_2 \leftarrow \{\{1\}, \{i_1, t_0\}, \{\}, \{\}\}$

$St.5: \; c_2 \leftarrow \{\{0, 1, 2\}, \{a_1, b_1, c_1, i_1, t_0\}, \{\}, \{\}\}$

10.4.4 Location creation in loops

In this section we discuss the creation of new locations in loop statements. Clearly, a class instance creation expression in the body of a loop statement means that in each iteration a new class instance is created at run-time. As we will see this poses no problem to the creation of the *ETFDM* of the loop, but has to be dealt with differently in the case of a purely static analysis like the *DFDM*.

To illustrate the main problems arising with new locations in loop bodies, look at method *while4(int i)* in Figure 10.9, which creates a linked list of the size of the parameter i, and then initializes it with newly created objects of type *obj*.

***ETFDM*:** As mentioned above when computing the *ETFDM* of a given loop statement we know the exact number of iterations performed at run-time, i.e., i. Assume that in each iteration an instance of class C is created through a class instance creation expression. We then also know that when the loop statement is executed, a total of i instances of class C is created. Since we are modeling each iteration separately, we automatically get i new locations representing the newly created objects. All FDs involving these new locations can be computed without any changes to the algorithms described above. If we look at the example we can distinguish multiple cases depending on the computed evaluation trace:

Case 1 ($i = 0$): In this special case the body of the loop is never executed. Therefore no new instances of class *obj* are created. No FDs arise from statement line 2. The return value l does not reference any new locations.

Case 2 ($i = 1$): Here the body of the **while** statement is evaluated exactly once. As a consequence, one instance of class *obj* is created, i.e., location 2. Variable l now references a list with one element, i.e., location 1, which, in turn, via *1.o* references the newly created instance of *obj*. The FDs arising from the loop statement in line 2 read as follows:

$\mathbf{ETFDM}^{(1)}_{\mathbf{while4(int\ i)}}$

$St.2: \ 2.value_1 \leftarrow \{\{0,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.2: \ 1.o_1 \leftarrow \{\{0\},\{l_1,i_0\},\{list.set(int,obj),obj.obj(int)\},\{2\}\}$

$St.2: \ i_1 \leftarrow \{\{0,1\},\{i_0\},\{\},\{\}\}$

Case 3 ($i = 5$): If the parameter i is set to five, a linked list with five elements is created. These elements are numbered locations 1 to 5. Before the loop statement is modeled, the linked list looks something like $l \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow null$. During the execution of the loop in line 2 another five class instances are created, i.e., instances of class *obj*. These instances are numbered location 6 to 10. After the execution of the loop all locations of type *obj* are referenced by elements of the linked list, i.e., $1.o \rightarrow 6$, $2.o \rightarrow 7$, etc... The FDs arising from the loop statement read as follows. Note that within the linked list each element depends on variable *next* of the locations of all previous elements, which can also be seen in the following FDs.

$\mathbf{ETFDM}^{(5)}_{\mathbf{while4(int\ i)}}$

$St.2: \ 6.value_1 \leftarrow \{\{0,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.2: \ 7.value_1 \leftarrow \{\{0,1,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.2: \ 8.value_1 \leftarrow \{\{0,1,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.2: \ 9.value_1 \leftarrow \{\{0,1,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.2: \ 10.value_1 \leftarrow \{\{0,1,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

$St.2: \ 1.o_1 \leftarrow \{\{0,1\},\{l_1,2.next_1,3.next_1,4.next_1,5.next_1,i_0\},$
$\{list.set(int,obj),obj.obj(int)\},\{6\}\}$

$St.2: \ 2.o_1 \leftarrow \{\{0,1\},\{l_1,3.next_1,4.next_1,5.next_1,i_0\},$
$\{list.set(int,obj),obj.obj(int)\},\{7\}\}$

$St.2: \ 3.o_1 \leftarrow \{\{0,1\},\{l_1,4.next_1,5.next_1,i_0\},\{list.set(int,obj),obj.obj(int)\},\{8\}\}$

$St.2: \ 4.o_1 \leftarrow \{\{0,1\},\{l_1,5.next_1,i_0\},\{list.set(int,obj),obj.obj(int)\},\{9\}\}$

$St.2: \ 5.o_1 \leftarrow \{\{0,1\},\{l_1,i_0\},\{list.set(int,obj),obj.obj(int)\},\{10\}\}$

$St.2: \; i_1 \leftarrow \{\{0,1\},\{i_0\},\{\},\{\}\}$

***DFDM*:** When we compute the *DFDM* of a given loop statement we do not know the number of iterations performed at run-time. This is why we compute the parameter *nestsize* (see Section 10.4.2) so that we can construct a nested **if** structure, whose FDs are equivalent to the FDs of the original loop statement (as far as all variable occurrences are concerned). If we do not adapt our modeling technique, we get *nestsize* new locations. In the general case this is not equivalent to the exact number of instances created at run-time.

A possible solution of this problem is to introduce the concept of multiple-locations. Since we do not know how many concrete locations to create at compile-time, we decide to create only one location, which is marked as multiple-location. The properties of multiple-locations are:

- Multiple-locations no longer stand for exactly one class instance created at run-time, but represent a set of locations of fixed type. They can be seen as abstract locations, which incorporate multiple possibly created instances in just one location. If we encounter a multiple-location, we cannot distinguish for which concrete locations the multiple-location has been created. All we know is that it represents $n \geq 0$ locations, which have been grouped together in the course of the modeling process.

- One example of the grouping of locations to multiple-locations is the modeling of loop statements. All locations created within the body of a loop statement by the same statement are part of the same multiple-location. Note that all these locations have the same type. Since we are not interested in the contents of the locations, we can make this abstraction without making the resulting model useless for the debugging process. Other examples of the usage of multiple-locations are presented in the following sections (see Chapters 11 and 13).

- So far we distinguished locations into explicitly created and default locations. Now we expand this concept by introducing multiple-new locations and multiple-default locations. An example of the use of multiple-default locations is given in the context of recursive methods (see Chapter 13).

Since multiple-locations have a different interpretation than concrete locations and represent a higher level of abstraction, we also have to treat them differently during the modeling process. In particular, if the content of a multiple location is changed, e.g., by assigning some value to one of its instance fields, we have to model the fact that the field of only one concrete class instance is modified and all other locations, which are also modeled by the multiple-location, remain unchanged. This can be done by introducing self dependencies similar to the ones introduced in Section 10.3.2. This aspect of the modeling process is demonstrated by the following example:

Example 10.4.4 *Consider an assignment statement of the form* `o.value = x;`, *where o is a variable referencing an instance of class obj (see Figure 9.2) and x is a variable of type* ***int****. If o represents a concrete location, say location 1, the resulting FD (without indices) is* $1.value \leftarrow x$. *If, on the other hand, location 1 is marked as a multiple-location, a self dependency has to be introduced leading to the FD* $1.value \leftarrow x, 1.value$. *The self dependency now models the fact that parts of the object space modeled by location 1 stay unchanged.*

Let us now look at the modeling of method *while4(int i)*. The call to method *create(int i)* in statement line 1 creates only one multiple-location, i.e., an imported new location 1, which has to be seen as an abstract representation of all elements of the linked list referenced by *l*. Interestingly, variable *1.next* now depends on location 1. This means that in contrast to all previously proposed models multiple-locations feature cyclic dependencies, which is due to their abstract representation of multiple class instances. In the loop statement in line 2 only one location is created, too. This multiple-new location stands for all class instances of type *obj*, which are possibly created at run-time. Note that whereas the number of created objects, i.e., loop iterations, is not known, the parameter *nestsize* is not used in this context, because it has nothing to do with the amount of locations created at run-time. The instance field *1.o* is then set to reference location 2, with both locations being marked as multiple. This means that the possibly created objects of type *obj*, which are summarized by location 2, are now referenced by the possibly created list elements, which are represented by location 1. Nevertheless, it is no longer possible to distinguish between individual list elements nor the objects referenced by these elements. The resulting *DFDM* looks as follows:

$\mathbf{DFDM_{while4(int\ i)}}$

$St.2: \ 2.value_1 \leftarrow \{\{0,1,100\},\{i_0\},\{obj.obj(int)\},\{\}\}$

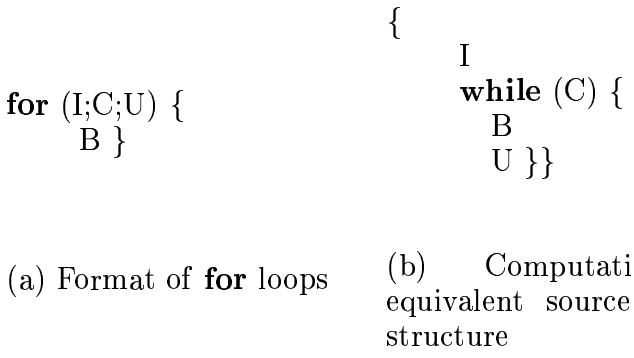
$St.2: \ 1.o_1 \leftarrow \{\{0,1\},\{l_1, 1.next_1, i_0\},\{list.set(int,obj), obj.obj(int)\},\{2\}\}$

$St.2: \ i_1 \leftarrow \{\{0,1\},\{i_0\},\{\},\{\}\}$

10.4.5 Modeling do and for statements

In previous sections we discussed the modeling of **while** statements in the context of *ETFDMs* and *DFDMs*. As already mentioned **do** and **for** statements can be modeled very similarly. Nevertheless, some details in the modeling process of these program components have to be considered separately, which is done in this section.

Do statements are very similar to **while** statements. The main difference is that the loop condition is at the end of the body in the case of **do** loops rather than at its beginning. This means that (1) there is no evaluation of the condition before entering the **do** statement and, as a consequence, (2) the loop body is always executed at least once.



(a) Format of **for** loops

(b) Computationally equivalent source code structure

Figure 10.10: Transforming a **for** loop into a **while** loop

When computing the *ETFDM* of a given **do** statement we apply the same techniques as for **while** statements, but simply do not consider the first condition. In other words, we create a nested **if** structure with $nestsize = i + 1$ and remove the loop condition from the outermost **if** statement making it a simple sub-block statement.

The same approach can be used to create *DFDMs* of **do** statements. Note that in the case of *DFDMs* the parameter *nestsize* has either to be computed or approximated using algorithms presented in Section 10.4.2.

For statements are a bit more complicated than **do** and **while** statements, because they include initializer and update blocks. Fortunately, **for** statements can be transformed into a computationally equivalent source code structure using only sub-blocks and **while** statements. Algorithm 10.4.2 describes the transformation of **for** loops:

Algorithm 10.4.2

1. Create a **while** loop with the same condition as the **for** statement. The body of the new **while** loop is the concatenation of (1) all statements of the body of the **for** statement and (2) all statement expressions of the **for** update block.

2. Create a `Java` block containing the concatenation of (1) all statement expressions of the **for** initializer block and (2) the **while** loop created in step 1.

Figure 10.10 (a) shows the general format of a **for** loop. The source code structure resulting from the application of Algorithm 10.4.2 is depicted in Figure 10.10 (b). Since the new source code structure is computationally equivalent to the original **for** statement, it can directly be used for the modeling process using only techniques already discussed. Note that this approach can be applied to both, the *ETFDM* and *DFDM*.

10.4.6 Model comparison

As demonstrated in the last couple of sections, the FDs arising from the body of a loop statement depend on the number of iterations performed at run-time. Moreover, FDs, which are correct for iteration i, might disappear in iteration $i+1$ and be replaced by new ones. The *ETFDM* computes all FDs arising in the context of a particular evaluation trace, i.e., for a certain number of iterations of a loop statement executed at run-time. The *DFDM*, on the other hand, does not make use of this dynamic information and aggregates all FDs arising from any iteration until all FDs are produced. Therefore, the number of FDs computed by the *ETFDM* is much smaller than the one in *DFDMs*, which is a huge advantage of the *ETFDM* in comparison to the *DFDM*. When we come to concrete debugging problems (see Part III) we find that potential diagnoses can so be eliminated much earlier in the debugging process due to the reduced set of FDs in the used model.

In the case of a location l being created in the loop statement's body, the *ETFDM* creates exactly one object per iteration, which leads to a total of i locations after i iterations. This approach is a very exact one, because it pretty much resembles the actual run-time behavior of a given `Java` system and comes close to a value-based modeling approach, which propagates actual values through the model. On the other hand the number of locations being created might be too large to make the resulting model understandable and suited for debugging. As shown, the *DFDM* creates exactly one location regardless of the number of iterations performed at run-time. This multiple-location has to be seen as an abstract representation of all locations possibly created during the execution of the loop statement. Whereas multiple-locations are much more abstract and cannot directly be mapped onto a single class instance created at run-time, the lower number of locations reduces the number of FDs in the model and, thus, speeds up the debugging process. Moreover, the specification of incorrect variables is so much easier for the user.

When computing the *ETFDM* there exists a different sub-trace for each loop iteration. This means that we also have to model the loop body i times and in case of hierarchical debugging store i models in the resulting diagnosis component. Clearly, the computational requirements of such an approach are much higher than the ones of the *DFDM*, where each sub-block of a statement is modeled and stored only once. Moreover, like with selection statements, the creation of *ETFDMs* requires much more memory than the computation of a static *DFDM*.

10.5 Synchronization statements

Software faults in multi-threaded `Java` systems, which stem from an incorrect synchronization between multiple threads, are out of the scope of this work. Note that other approaches have been designed to specifically deal with the localization of these faults (see [12]).

Herein we assume that no faults of this sort arise. We therefore simply ignore the **synchronize** keyword and model the **synchronize** statement's expression

and its sub-block one after another.

> *JADE: The JADE system currently ignores the **synchronize** keyword during the computation of both models, the ETFDM and DFDM. Note that in either case no run-time information is used.*

10.6 Try statements

When computing the *ETFDM* of a **try** statement, once again all run-time information is present so that we can collect all FDs arising from the whole **try** statement.

This, however, is not so straightforward in the case of *DFDMs*. A simple solution is to ignore all exceptions and assume that each **try** block in the `Java` system terminates without an exception being thrown. We could then simply model the main block of the statement followed by the **finally** block, if any. Clauses do not have to be modeled following the assumptions made.

Another approach is to construct all FDs possibly arising from all run-time scenarios. This includes all potential exceptions being thrown at run-time. Clearly, this approach is rather complex and results in a high amount of FDs being created. However, in this work we forgo to dicuss the exact modeling of **try** statements in more detail.

> *JADE: The JADE system currently ignores the concept of exceptions at run-time and relies on all **try** blocks to terminate normally. Therefore, only the **try** block is modeled and all catch clauses are ignored. In case of a **finally** block this block is modeled after the main block of the **try** statement.*

Chapter 11

Modeling Arrays & Strings

Like most imperative and object-oriented programming languages `Java` features the creation and use of multidimensional arrays and strings. In this section we shortly describe `Java` arrays and strings and show how they can be incorporated into the various FDMs.

11.1 `Java` arrays

`Java` arrays are dynamically created objects of array type and may be assigned to variables of type *Object* (see [16]). Each array object has a non-negative number of components, which do not have names, but can be accessed through array access expressions. If an array has n components, the array is said to have a length of n. All components of an array must have the same type, i.e., the component type. A component type may itself be a `Java` array. The innermost component type of a (possibly multidimensional) array must have a type other than array type. This type is called the element type of the original array. The components at this level are called elements of the original array.

There are multiple ways of declaring a variable of array type. However, such a declaration does not create an array object nor allocates space for array components. It only creates the variable itself, which can contain a reference to an array. There are two ways of creating an array object:

- An array creation expression creates an array object by specifying the element type, the number of levels of nested arrays, and the length of the array for at least one of the levels of nested arrays.
- An array initializer creates an array by providing initial values for all its components. The length of the constructed array is equal to the number of expressions in the array initializer. Multidimensional arrays may be created using nested array initializers, whose elements again are array initializers.

Components of an array may be accessed by an array access expression, which consists of an array reference and an indexing expression enclosed by '[' and ']'. Usually the length of an array can be accessed through the final instance variable *length*.

```
        void array1(); {
          ...
n.        int[] a = {10,20,30,40,50};
          ...
m.        int i = a[3];
          ...
        }
```

Figure 11.1: Example method *array1()*

Example 11.1.1 *Figure 11.1 shows a Java source code fragment, which includes an array initializer in statement line n. The resulting array a has one dimension with five components of type* ***int****. Thus the length of a is five. The components of a are also its elements, because there appear no nested arrays in a.*

11.2 Modeling arrays

***ETFDM*:** When using an evaluation trace, we can always determine not only the number of dimensions of a given array a, but also the length of a. Each time a component of a (no matter whether it is an element of a or not) is accessed through an array access expression, we can determine its exact index as the evaluation value of the indexing expression. Therefore, when building the *ETFDM* of an array a, we can exactly model a by assigning it a new location l. Furthermore, all components of a can be represented separately as variables $l.element\_1$ to $l.element\_n$ with n being the length of array a. The length of a itself can also be modeled by introducing a new FD $l.\_length \leftarrow n$.

Example 11.2.1 *If we look at the example in Figure 11.1 we find an array initializer in statement line n. The resulting ETFDM (note that in this case statement line n does not depend on an evaluation trace) reads as follows:*

$\mathbf{ETFDM}^{()}_{array1()}$

$St.n: \ 1.element\_1_1 \leftarrow \{\{10\},\{\},\{\},\{\}\}$

$St.n: \ 1.element\_2_1 \leftarrow \{\{20\},\{\},\{\},\{\}\}$

$St.n: \ 1.element\_3_1 \leftarrow \{\{30\},\{\},\{\},\{\}\}$

$St.n: \ 1.element\_4_1 \leftarrow \{\{40\},\{\},\{\},\{\}\}$

$St.n: \ 1.element\_5_1 \leftarrow \{\{50\},\{\},\{\},\{\}\}$

$St.n: \ 1.length \leftarrow \{\{5\},\{\},\{\},\{\}\}$

$St.n: \ a_1 \leftarrow \{\{\},\{\},\{\},\{1\}\}$

Example 11.2.2 *If we now look at the array access expression in the assignment in line m of Figure 11.1, we get the following FDs. Note that j and k stand for the current indices of a and element_3, respectively.*

$\mathbf{ETFDM}^{()}_{array1()}$

$St.m:\ i_1 \leftarrow \{\{3\}, \{a_j, 1.element\_3_k\}, \{\}, \{\}\}$

Clearly, the resulting model is very precise, but also introduces a lot of new variables, i.e., array components, into the model.

> *JADE: The JADE system currently does not support the modeling of arrays with run-time information. Therefore, the DFDM modeling component, which is presented below, is used in all cases.*

***DFDM*:** The problem during the computation of the *DFDM* is that at compile-time we can only determine the length of array a, but not the exact component addressed by an individual array access expression. We therefore introduce a new level of abstraction by grouping all components to a new abstract variable *element*. This variable now stands for all components or elements of array a. All language structures influencing any of the components of a are collected in a single DEP structure, on which the abstract variable *element* finally depends. The resulting model is not as exact as the one making use of run-time information. It contains exactly one variable for each array location (apart from variable *length*) and is so easier to read and smaller in its amount of FDs. Note that the more abstract representation of array elements is similar to the concept of multiple-locations (see Section 10.4).

Example 11.2.3 *If we, again, look at the array initializer in the example in Figure 11.1, we can now compute the DFDM of method array1(). As we do not distinguish the various components of array a we get the following FDM:*

$\mathbf{DFDM}_{array2()}$

$St.n:\ 1.element_1 \leftarrow \{\{10, 20, 30, 40, 50\}, \{\}, \{\}, \{\}\}$

$St.n:\ 1.length \leftarrow \{\{5\}, \{\}, \{\}, \{\}\}$

$St.n:\ a_1 \leftarrow \{\{\}, \{\}, \{\}, \{1\}\}$

Example 11.2.4 *The array access expression in line m is now associated with any of the components of a. Note that in this case the indexing expression could be resolved at compile-time, which is generally not the case. The resulting FDM for statement line m looks as follows:*

```
    int array2() {
1.    arrayTest o = new arrayTest();
2.    int i[] = {1,2,3,4,5};
3.    int j[] = new int[] {10,11,12};
4.    arrayTest a[][] = new arrayTest[][] {{o,null},new arrayTest[]{null,o}};
5.    return i[3]+j[2];
    }
```

Figure 11.2: Example method *array2()*

$\mathbf{DFDM}_{array2()}$

$St.m: \; i_1 \leftarrow \{\{3\}, \{a_j, 1.element_k\}, \{\}, \{\}\}$

Note that, again, j and k stand for the current index of a and *1.element*, respectively.

11.3 An example *DFDM*

In order to further highlight the modeling principles of arrays consider method *array2()* of class *arrayTest*, which is depicted in Figure 11.2. The full *DFDM* arising from *array2()* reads as follows:

$\mathbf{DFDM}_{\mathbf{array2()}}$

$St.1: \; o_1 \leftarrow \{\{\}, \{\}, \{arrayTest.arrayTest()\}, \{1\}\}$

$St.2: \; 2.\_arrayLength_1 \leftarrow \{\{5\}, \{\}, \{\}, \{\}\}$

$St.2: \; 2.\_array_1 \leftarrow \{\{1,2,3,4,5\}, \{\}, \{\}, \{\}\}$

$St.2: \; i_1 \leftarrow \{\{1,2,3,4,5\}, \{\}, \{\}, \{2\}\}$

$St.3: \; 3.\_arrayLength_1 \leftarrow \{\{3\}, \{\}, \{\}, \{\}\}$

$St.3: \; 3.\_array_1 \leftarrow \{\{10,11,12\}, \{\}, \{\}, \{\}\}$

$St.3: \; j_1 \leftarrow \{\{10,11,12\}, \{\}, \{\}, \{3\}\}$

$St.4: \; 4.\_arrayLength_1 \leftarrow \{\{2\}, \{\}, \{\}, \{\}\}$

$St.4: \; 4.\_array_1 \leftarrow \{\{null\}, \{o_1\}, \{\}, \{5,6\}\}$

$St.4: \; 5.\_arrayLength_1 \leftarrow \{\{2\}, \{\}, \{\}, \{\}\}$

$St.4: \; 5.\_array_1 \leftarrow \{\{null\}, \{o_1\}, \{\}, \{1\}\}$

$St.4: \; 6.\_arrayLength_1 \leftarrow \{\{2\}, \{\}, \{\}, \{\}\}$

$St.4: \; 6.\_array_1 \leftarrow \{\{null\}, \{o_1\}, \{\}, \{1\}\}$

$St.4: \; a_1 \leftarrow \{\{null\}, \{\}, \{\}, \{4\}\}$

$St.5: \ \_result_1 \leftarrow \{\{2,3\},\{i_1,j_1,2.\_array_1,3.\_array_1\},\{\},\{\}\}$

11.4 Modeling Java strings

In Java strings are implemented as instances of the system class *String*. This class is part of the standard package *java.lang* and thus accessible from every Java host environment. Due to the importance of strings, instances of class *String* differ from other class instances in a couple of peculiarities, some of which are shortly described in the following list. For a more detailed description of Java strings refer to the Java Language Specification [16].

- Strings cannot only be created through explicit constructor calls, but also by the specification of a string literal, e.g., "Jade", which at run-time is automatically promoted to an instance of class *String*.
- Instances of class *String* always have a constant value, i.e., their contents cannot be changed.
- Instances of class *String* can be interned. This means that they are added to an internal data structure kept by class *String*, where each string can be stored only once. When different instances of class *String* with the same contents are interned, they become a single instance stored in the internal data structure. Note that all strings created by string literals are automatically interned.
- If one argument of the + operator is of type *String*, the Java string conversion rules apply. This means that in this special case the other argument to the + is converted to a *String*, and a new string object which is the concatenation of the two strings is the result of the + operation.

Since all system classes can be modeled exactly like user-defined classes (see Section 12.4), class *String* can be modeled by using techniques described in previous chapters. However, due to the special properties of strings some additional features have to be added to the modeling component. The exact modeling of Java strings is beyond the scope of this work. The following list shows some points, which have to be considered during the modeling of strings:

- String literals have automatically to be converted to instances of class *String* by the modeling component. This includes creating a new location of type *String*.
- Similar techniques have to be applied in the case of string conversions.
- The internal data structure keeping track of interned string objects has to be explicitly modeled by the modeling component. This means that all interned strings with the same contents have to be represented by the same location rather than different ones. This includes a special handling of method *intern()* of class *String*.

JADE: *The* ***JADE*** *system currently does not support the special properties of strings as discussed above. This means that generally string objects can be used in the* ***JADE*** *debugging environment, but it can currently not be guaranteed that the correct FDs are computed in case of string literals, interned strings, and string conversions.*

Chapter 12

Modeling Methods

In previous chapters we have described how *ETFDMs* and *DFDMs* can be constructed at expression and statement level. In this section we show how individual statement models can be used to create a complete FD method model, which in turn serves as a constituent of FD class, package or host environment models.

12.1 FD method models

As defined in Chapter 6 the FD method model of method m, i.e., FDM_m, is a collection comprising the FDs of all FD statement models of method m, i.e., $\{FDM_s \mid s \in m\}$. Although the chronological order of these FDs is implicitly given by the variable indices, we herein assume that all FDs are ordered by their statement indices.

Apart from the FDs there are other parts of a FD method model, which are either used during the modeling process or may be needed after completion of the modeling process in addition to the resulting FDs. These parts are shortly described in the following list:

The modeled method is the source code of the `Java` method that serves as the source system during the modeling process.

Diagnosis components are the individual statements of the modeled method together with all FDs arising from these statements. In case of statements, which contain sub-blocks (e.g., loop or selection statements), the FDMs of all sub-blocks are stored in the (hierarchical) diagnosis components as well.

The summarized model, i.e., $sum(FDM_m)$, can be computed using Algorithm 6.6.2 proposed in Section 6.6. The summary of method m can either be computed on demand or be created once and stored with FDM_m. In the case of *DFDMs* where the same summarized FDM is used to model multiple method calls, the latter approach is probably superior.

The locations are created in the course of modeling m. This includes all types of locations, i.e., new, default, and imported locations. Locations are not

only needed during the modeling process, but store important information about a location's type and state (e.g., single- or multiple-locations).

The variable environment stores all variables together with their indices. In case of a variable v of reference type all locations, which v currently references (*ETFDM*) or possibly references (*DFDM*), are part of the variable environment, too. The variable environment is mainly used during the modeling process to keep track of variable indices and object references. In the latter functioning the variable environment is the key component allowing for the handling of aliasing.

The Evaluation trace of m has to be stored together with FDM_m in case of the *ETFDM*. Note that a certain *ETFDM* is only correct for method m in the context of a specific evaluation trace and cannot be used in the general case.

Example 12.1.1 *Let us come back to method test() from Section 5.2 (see Figure 5.1). A full FD method model (DFDM) of method test() can be represented as follows:*

(1) Modeled method: *see Figure 5.1*

(2) Diagnosis components: *All statements of method test(), i.e.,* $\{s_1, s_2, s_3, s_4, s_5\}$*, together with the following associated FDs:*

DFDM$_{test()}$

$St.1: 1.x_1 \leftarrow \{\{0\},\{\},\{Point(int,int)\},\{\}\}$

$St.1: 1.y_1 \leftarrow \{\{0\},\{\},\{Point(int,int)\},\{\}\}$

$St.1: p1_1 \leftarrow \{\{\},\{\},\{Point(int,int)\},\{1\}\}$

$St.2: 2.x_1 \leftarrow \{\{2\},\{\},\{Point(int,int)\},\{\}\}$

$St.2: 2.y_1 \leftarrow \{\{3\},\{\},\{Point(int,int)\},\{\}\}$

$St.2: p2_1 \leftarrow \{\{\},\{\},\{Point(int,int)\},\{2\}\}$

$St.3: 1.x_2 \leftarrow \{\{1\},\{p1_1\},\{\},\{\}\}$

$St.4: 1.y_2 \leftarrow \{\{2\},\{p1_1\},\{\},\{\}\}$

$St.5: 3.x_1 \leftarrow \{\{\},\{p1_1,p2_1,1.x_2,2.x_1\},\{Point(int,int),plus(Point)\},\{\}\}$

$St.5: 3.y_1 \leftarrow \{\{\},\{p1_1,p2_1,1.y_2,2.y_1\},\{Point(int,int),plus(Point)\},\{\}\}$

$St.5: p2_2 \leftarrow \{\{\},\{p1_1\},\{Point(int,int),plus(Point)\},\{3\}\}$

Note that in this case no sub-block models exist, because test() does not contain any loop or selection statements.

Var	i	locs
1.x	2	{}
1.y	2	{}
2.x	1	{}
2.y	1	{}
3.x	1	{}
3.y	1	{}
p1	1	{1}
p2	2	{3}

Figure 12.1: Variable environment of method *test()*

(3) Summarized model:

$\mathbf{sum}(\mathbf{FDM}_{test()})$

$1.x \leftarrow \{\{1\}, \{\}, \{Point(int, int)\}, \{\}\}$

$1.y \leftarrow \{\{2\}, \{\}, \{Point(int, int)\}, \{\}\}$

$2.x \leftarrow \{\{2\}, \{\}, \{Point(int, int)\}, \{\}\}$

$2.y \leftarrow \{\{3\}, \{\}, \{Point(int, int)\}, \{\}\}$

$3.x \leftarrow \{\{1, 2\}, \{\}, \{Point(int, int), plus(Point)\}, \{\}\}$

$3.y \leftarrow \{\{2, 3\}, \{\}, \{Point(int, int), plus(Point)\}, \{\}\}$

$p1 \leftarrow \{\{\}, \{\}, \{Point(int, int)\}, \{1\}\}$

$p2 \leftarrow \{\{\}, \{\}, \{Point(int, int), plus(Point)\}, \{3\}\}$

(4) Locations:

- *New location 1 of type Point*
- *New location 2 of type Point*
- *Imported location 3 of type Point*

(5) Variable environment: *see Figure 12.1*

(6) Evaluation trace: *none (DFDM)*

12.2 Using default models

In some cases a FDM of method m cannot be constructed, what leads to an incomplete model of the whole `Java` host environment. This can happen under the following circumstances:

- The source code of m is not fully available. This can be the case, if m is defined as a native function, mostly as part of some `Java` system class, or if we are dealing with an incomplete source code system, which only defines the signature of method m and assumes that its byte-code is linked to the system at run-time.
- Method m cannot be modeled, because it contains unsupported language structures, e.g., exception handling, **break** and **continue** statements, etc...

In both cases we are interested in a default model of method m, which does not have to be a precise model of the internal structure of m, but should enable the modeling system to model methods containing calls to method m. The primary goal of a default model of method m is therefore not a detailed model of m itself, but a summarized model of m that guarantees that a given `Java` host environment can be modeled completely, including models of all system classes and methods.

When creating a default model of m, we look at m as a black box and consider all variables, which might change their values during the execution of m. These output variables are:

- All instance fields of the receiver
- All instance fields of objects, which can be accessed from the receiver through instance fields, static variables, or arguments.
- All static variables of all classes of the system, which can be accessed through the receiver or objects accessed by the receiver.
- The return value of m

On the other hand we have to determine all variables, which possibly influence the variable changes assumed above. These input variables are:

- All instance fields of the receiver
- All instance fields of objects, which can be accessed from the receiver through instance fields, static variables, or arguments.
- All static variables of all classes of the system, which can be accessed through the receiver or objects accessed by the receiver.
- All arguments of m

We can now easily create a (summarized) default FDM of method m by assuming that all output variables of m depend on all input variables of m. Clearly, this approach results in a very complex model of m including too many FDs. This model cannot be used for the debugging of m. On the other hand we make sure that for all methods of a given system at least a default model exists. We therefore can model all methods including method calls to m.

Another approach is to let the programmer specify a set of intended FDs for a certain method during the software development process. This is a very difficult task and it is doubtable whether the majority of programmers would do that, but it could in the end turn out as a very powerful approach towards an automatic fault localization tool.

12.3 FD class, package, and host environment models

A static FD class model theoretically includes the FDMs of all class body declarations. This includes not only the models of all method and constructor declarations, but also the FDs arising from assignments within field declarations and the FD block models computed for static initializers. Note that in case of the *ETFDM*, which is computed for a single evaluation trace, it does not make sense to compute a class model, because such a model would only be correct in the context of a certain run-time behavior of m. This is also true for FD package and host environment models.

> ***JADE:*** *The* `JADE` *system currently only models method and constructor declarations. This means that during the debugging process source code faults in field declarations and static initializers cannot be found.*

FD package models simply consist of the class models of all classes within the package. FD host environment models store all package models currently loaded into the system.

12.4 Modeling system classes

Since all `Java` systems include at least some of the pre-defined system classes, an efficient modeling of these classes or packages has to be guaranteed. Generally, system classes can be modeled like all other classes, because they normally include standard `Java` methods. Note that native methods can be modeled by creating default methods (see Section 12.2).

When computing the *ETFDM* of method m, which calls a system method s, method s has to be modeled with a concrete evaluation trace on demand. Note that default models could be used in the context of *ETFDMs*, but this would nullify the advantages gained from *ETFDMs*.

When using *DFDMs* the models of all system classes can be computed off-line and stored in a separate database. During the modeling process the models of system classes can then be loaded into the system, whenever a method call to a system method appears in the currently modeled method. This approach requires a larger amount of system memory, but can avoid a repeated modeling of system classes and thus speed up the modeling process.

Chapter 13

Handling Recursion

So far we have only considered the modeling of non-recursive methods. In the following sections we enhance the modeling process in a way that it is able to handle directly and indirectly recursive method calls.

13.1 Introduction

In Section 9.2 we show how to model a method call of method n appearing in method m. When computing the *ETFDM* of such a method call, the FDM of the called method has to be computed on demand for the method call's sub-trace. Existing models cannot be reused due to the dynamic character of the *ETFDM*. As a consequence, recursive method calls can be treated exactly like all other, i.e., non-recursive, method calls. The termination of the modeling algorithm can be guaranteed due to the finite evaluation trace. Furthermore, it is one of our initial assumptions that only programs, which can be shown to terminate on their inputs, are considered for modeling (see Section 5.2).

When, on the other hand, we compute the *DFDM* of such a method call, we have to rely on the existence of a FD method model of the called method n. There are generally two different scenarios:

1. Method n has already been modeled and stored in the current FD host environment model. In this case the existing model can directly be used to perform the modeling of the method call.

2. FDM_n has not been computed yet. In this case the modeling of method m has to be suspended in order to first compute the FDM of method n. Only if the latter model is present the modeling of method m can be resumed.

The problem is hidden in the second case. Clearly, such a strategy can only succeed, if method n and all methods called by n do not, in turn, call method m. We say, that method n must not be recursive with method m.

Definition 13.1.1 *Two methods, m and n, are said to be recursive with each other, if m or one of the methods called by m calls method n and, in turn, n*

```
interface Queen {
      boolean first();
      boolean next();
      boolean checkRow(int r, int c);
      void result(int[] r);
}
```

Figure 13.1: Example interface *Queen*

```
class NullQueen implements Queen {
  public boolean first() {
1.      return true; }
  public boolean next() {
1.      return false; }
  public boolean checkRow(int r, int c) {
1.      return false; }
  public void result(int[] r) {
}
```

Figure 13.2: Example class *NullQueen*

or one of the methods called by n calls method m. If $m = n$ we call it a direct recursion. If $m \neq n$ we speak about an indirect recursion.

Figures 13.1 to 13.4 show a `Java` implementation of the N-Queens-problem. It has been included in this chapter, because it will help demonstrating some important issues of recursive FD modeling. At this stage we note that most methods presented in Figure 13.3 are recursive ones. Whereas method *checkRow(int r, int c)* of class *ConcreteQueen* is directly recursive, methods *testPosition()*, *next()*, and *advance()* of class *ConcreteQueen* are indirectly recursive with each other.

It is obvious that a method call of method n, which is recursive with the calling method m, results in an infinite modeling loop. While method m tries to compute the FDM of n first, method n relies on the existence of a model of method m. The modeling process will eventually encounter an infinite loop.

We therefore have to enhance the modeling process for the *DFDM* in order to handle recursive methods. The following sections present a fix-point algorithm, which computes the FDMs of recursive methods by successively adding FDs to a method's model until a stable model is found.

Note that all new concepts introduced in this section do not change the modeling approach of non-recursive methods used so far. Furthermore, the non-recursive modeling can be regarded as a special case of the modeling of recursive methods. The algorithms proposed in the following sections will therefore work for both, the recursive and non-recursive case.

```
      class ConcreteQueen implements Queen {
        int row, col;
        Queen neighbor;
        ConcreteQueen(int c, Queen n) {
1.        col = c;
2.        neighbor = n; }
        public boolean first() {
1.        neighbor.first();
2.        row = 1;
3.        return testPosition(); }
        public boolean next() {
1.        boolean success = advance();
2.        if (success) success = testPosition();
3.        return success; }
        public boolean checkRow(int r, int c) {
1.        int colDiff = c - col;
2.        return row == r
            || row + colDiff == r
            || row - colDiff == r
            || neighbor.checkRow(r,c); }
        boolean testPosition() {
1.        boolean success = true;
2.        while (success && neighbor.checkRow(row, col))
2.1.        success = advance();
3.        return success; }
        boolean advance() {
1.        boolean success = true;
2.        if (row == 8) {
2.1.        success = neighbor.next();
2.2.        if (success) row = 1; }
2.3.      else ++row;
3.        return success; }
        public void result(int[] r) {
1.        r[col] = row;
2.        neighbor.result(r); }
      }
```

Figure 13.3: Example class *ConcreteQueen*

13.2 The method dependency graph

In order to find out, whether there are any recursive method calls in a given `Java` system and to determine the modeling sequence of multiple methods we construct a method dependency graph (MDG). A MDG is a directed graph (see Section 6.6) with cycles or loops in the case of recursive method calls. More formally, a MDG is defined as follows:

```
      class EightQueens {
        static void demo() {
1.        Queen qs = new NullQueen();
2.        for(int i = 8; i > 0; i − −)
2.1.        qs = new ConcreteQueen(i,qs);
3.        qs.first();
4.        int[] result = new int[9];
5.        qs.result(result);
        }
      }
```

Figure 13.4: Example class *EightQueens*

The vertices V: The vertices of a MDG are the methods of the `Java` system. There exists a direct correspondence between the vertices of the MDG and the methods of the `Java` program. Each vertex of the MDG is labeled by the signature of the corresponding method of the `Java` program. Note that each method of the program is uniquely identified by its signature, which consists of the name of the class owning the method (note that different classes may have methods with the same name) followed by the name of the method and the types of its formal parameters (a class may have different methods with the same name but different types and/or numbers of parameters).

The edges E: There exists a directed edge in the MDG from a vertex m to a vertex n iff there exists at least one statement in the body of method m containing a method call to method n.

Figure 13.5 shows the MDG of class *ConcreteQueen* of our example program in Figure 13.3. Note that for readability purposes all methods are depicted only with their simple names (i.e., not prefixed with their class names). Cycles in the MDP mean that we have to deal with recursive methods. Moreover, all methods lying on a cycle are recursive with each other.

Building on the MDG we can now compute all sets of mutually recursive methods, i.e., a maximum number of sets, which contain only methods, which are recursive with one another. This can easily be achieved by computing all strongly connected components (SCCs) of the MDG. Algorithms for efficient computations of a directed graph's SCCs are presented in [45] and [35].

Definition 13.2.1 *According to [35] two nodes v and w in a graph G are path equivalent if there is a path from v to w and a path from w to v. Path equivalence partitions V into maximal disjoint sets of path equivalent nodes. These sets are called the strongly connected components (SCCs) of the graph G.*

Definition 13.2.2 *A SCC is called recursive, if it contains one or more recursive methods. Note that all SCCs with two or more methods are always recursive SCCs.*

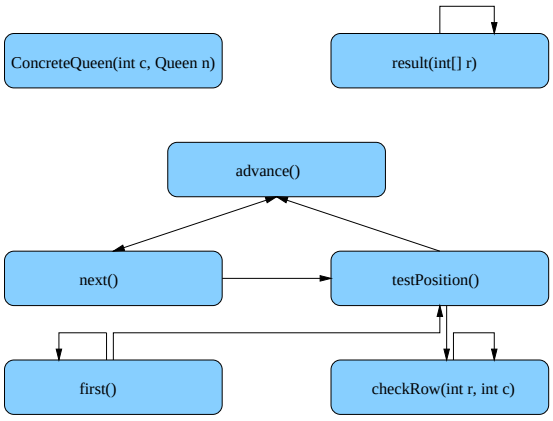


Figure 13.5: MDG of class *ConcreteQueen*

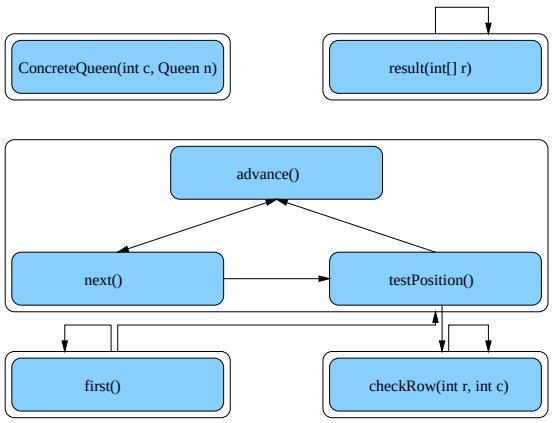


Figure 13.6: SCCs of class *ConcreteQueen*

The SCCs of class *ConcreteQueen* of our example program are depicted in Figure 13.6. We see that now the three methods *advance()*, *next()*, and *testPosition()*, which are recursive with one another, all lie in the same SCC. All other methods are either non-recursive or directly recursive. Therefore, they constitute SCCs on their own.

Note that non-recursive methods always constitute SCCs themselves. All `Java` examples presented in previous sections can therefore easily be transformed into their SCC representation by creating exactly one SCC for each method of the `Java` system. As we saw the modeling of these SCCs (i.e., SCCs containing only one non-recursive method) is not a problem and can be performed using the algorithms presented so far. Only recursive SCCs need to be treated separately.

Thus the new modeling approach, when creating the *DFDM* of a given `Java` system, is to model all SCCs one after another. Again, if during the modeling of one SCC models of methods in other SCCs are needed, the modeling process has to be suspended, the other SCC has to be computed first, and finally the

modeling of the original SCC can be resumed. Due to the acyclic structure of the graph created by the SCCs of a `Java` program, we can be sure that this process will terminate. The only problem remaining is how to model recursive SCCs. This can be done by applying a fix-point computation, which will be the main topic of the next couple of sections.

13.3 Top-level detection of infinite recursions

The problem with the modeling of recursive methods is that we eventually encounter an infinite modeling sequence during the modeling of recursive method calls. In the following we will distinguish between method calls, which are definitely executed at run-time, and method calls, whose execution depends on a concrete evaluation trace. In order to do so we define a function $isRecursiveAtTopLevel(m)$, which determines, whether method m contains recursive method calls, which are executed regardless of the used evaluation trace. More formally, we write:

Definition 13.3.1 *Let b be the outermost block of method m. Then the following equation holds: $isRecursiveAtTopLevel(m) = isRecursiveAtTopLevel(b)$, where $isRecursiveAtTopLevel(b)$ returns true iff:*

- *there appears a recursive method call in b*
- *b contains a selection statement s so that $isRecursiveAtTopLevel(h)$ holds for all branches h of s.*
- *b contains a **do** statement s so that $isRecursiveAtTopLevel(h)$ holds for the body h of s*

Note that recursive method calls in the bodies of **for** and **while** statements are not considered here, because in contrast to **do** loops the bodies of these statements are not necessarily executed.

Example 13.3.1 *Let us now come back to the methods defined for class ConcreteQueen (see Figure 13.3). As already mentioned the methods advance(), next(), and testPosition() lie in the same SCC, which has been depicted in Figure 13.6. Method next(), for instance, contains a recursive method call, i.e., a call to method advance(), in statement line 1. Clearly, this statement is executed in all possible evaluation traces, which means that $isRecursiveAtTopLevel(next()) = true$ holds. If we now look at the methods advance() and testPosition(), we find recursive method calls only in a **while** and an **if** statement (with an empty else-branch), respectively. This means that at run-time the recursive method call is not always executed and, therefore, $isRecursiveAtTopLevel(advance()) = false$ and $isRecursiveAtTopLevel(testPosition()) = false$ hold.*

Building on function $isRecursiveAtTopLevel(m)$ we can now distinguish the following two cases:

Case 1: Assume all methods of a certain SCC contain recursive method calls, which are definitely executed at run-time, i.e. $isRecursiveAtTopLevel(m) = true \ \forall m \in SCC$. In this case we expect the method to encounter an infinite sequence of recursive method calls, leading to the non-termination of the whole program. In most cases the reason for this behavior is a missing exit condition of at least one of the recursive methods. As stated in Section 5.2 we are only interested in the modeling of `Java` programs, which terminate on all inputs. This explicitly excludes the modeling of infinite loops, method call sequences, and run-time failures. We detect such a program behavior at compile-time by appropriately modifying Tarjan's algorithm. Instead of modeling the system, a warning is displayed to the user.

Case 2: There exists at least one method in a certain SCC, which does not contain recursive method calls that are definitely executed, i.e., $\exists m \in SCC \mid isRecursiveAtTopLevel(m) = false$. In this case we will start our fix-point iteration modeling process by first modeling the top-level structure of m. The detailed algorithm will be presented in the following sections.

13.3.1 Polymorphism and recursion

As mentioned in Section 9.2, when computing the *DFDM* of a polymorphic method call, we generally have to consider the FDMs of all methods, which are possibly called at run-time. Of course, the same is true for calls to polymorphic recursive methods. Consider a method call, which at run-time may call one of the polymorphic methods n_1, n_2, or n_3. Let further method m contain the method call at its top-level. The function $isRecursiveAtTopLevel(m)$ evaluates to `true` iff method m is recursive with, i.e., in the same SCC as, all three possibly called methods. In other words, if there exists one non-recursive method, which is possibly called at run-time, then the whole method call has to be regarded as non-recursive as far as function $isRecursiveAtTopLevel(m)$ is concerned.

Example 13.3.2 *If we, again, look at method advance() of class ConcreteQueen (see Figure 13.3), we find a polymorphic recursive method call to method next() in statement line 2.1, i.e., in the then-branch t of the* ***if*** *statement in line 1. This method call has to be seen as polymorphic, because at compile-time it is not known, whether the receiver (referenced by variable neighbor) is of type ConcreteQueen or of type NullQueen (see Figure 13.2). Because of the fact that method next() of class NullQueen is not recursive,* $isRecursiveAtTopLevel(t) = false$ *holds, although method next() of class ConcreteQueen is recursive.*

13.3.2 Evaluation of expressions

Unfortunately the algorithm to compute $isRecursiveAtTopLevel(m)$ as introduced above does not always produce the right result. One special case, which should be mentioned here for the sake of completeness, are conditional expressions, which are not always fully evaluated during run-time. Method *check-*

Row(int r, int c) in Figure 13.3, for instance, contains a top-level recursive method call to itself. This leads to the assumption at compile-time that we necessarily encounter an infinite method call sequence at run-time. This assumption, however, is not true, because under certain conditions, i.e., when one of the first three sub-expressions of the return expression evaluate to `true`, the rightmost sub-expression, i.e., the recursive method call, is not executed.

> ***JADE:*** *Currently, the* ***JADE*** *system cannot handle the special case described above. This means that an infinite method call sequence will be assumed by the system, what causes the modeling process to be terminated. Note that in the above example the correct FDM can only be computed due to the polymorphism of the method call.*

13.4 Fix-point iteration modeling

We now want to compute the FDM of a recursive SCC, i.e., the model of all methods within the SCC. As shown above these methods have to be recursive with one another and can therefore not be modeled separately. Moreover, all methods of a given SCC have to be modeled together, which can be done by employing a fix-point computation algorithm.

The fix-point algorithm starts with computing an initial model of the root method of a given SCC. Generally, the root method is the method in a SCC, which is visited first by Tarjan's algorithm (see [45]). It has no special properties apart from the fact, that SCC roots are always accessible from outside the SCC, i.e., root methods are always methods, which are called by methods in other SCCs. The initial root model can then be used to create initial models of further methods in the SCC, which in turn allow for the creation of other methods etc... Eventually, there should exist initial models for all methods within a given SCC. As we will see in Section 13.4.1 this approach succeeds provided that no infinite recursion is detected during the creation of the SCC graph.

The fix-point algorithm then uses the initial method models to compute more detailed models in its next iteration. Generally speaking, all methods of a SCC of iteration i are needed in order to compute the model of the complete SCC in iteration $i+1$. This process is repeated until an iteration t is reached, in which the models of all methods are equivalent to the respective models in iteration $t-1$. In this case we say that the fix-point of the modeling algorithm is reached.

Note that if during the computation of the models of the methods from a SCC calls to methods belonging to another SCC' are encountered, the computation of the models of the methods from SCC is suspended and the models of the methods from SCC' are computed by employing the same algorithm as for SCC. This approach does not lead to infinite loops because each cycle in a MDG is covered by a single SCC, i.e., all edges connecting two distinct SCCs have the same direction.

13.4.1 Initialization iteration

The first problem one has to deal with in any fix-point algorithm is how to initialize the main data structures. When applying a fix-point modeling algorithm, we are looking for initial FDMs of all methods of a certain SCC. These models are subsequently used as the basis for future computations. The initial method models can be constructed using Algorithm 13.4.1:

Algorithm 13.4.1

Step I: The pre-condition for the fix-point algorithm as discussed in Section 13.3 is that there exists at least one method in a SCC, which does not contain recursive method calls, which are definitely executed, i.e., $\exists\, m \in SCC \mid isRecursiveAtTopLevel(m) = false$. If this condition is satisfied we start the fix-point algorithm by computing an initial model for method m. This can be done by modeling only the method's top-level block (which is guaranteed to contain no recursive method calls) and all sub-blocks, which do not contain method calls to other methods within the same SCC. This means that during the initialization phase all sub-blocks containing recursive method calls are ignored. The resulting model of method m contains only correct FDs, but generally some FDs are missing. It is the goal of the fix-point algorithm to add these FDs in the following iterations. Note that there might as well exist other methods m', which do not contain top-level recursive method calls, i.e., for which the condition $isRecursiveAtTopLevel(m') = false$ holds. All these methods can immediately be modeled in the same way as method m.

Step II: We then choose a method n from the SCC with the following properties:

- The initial model of method n has not yet been computed.
- The initial models of all methods in the same SCC as n, which are called at method n's top-level, have been computed.

Note that a method n meeting these criteria can always be found, if no top-level, recursive method call sequence in the SCC exists. However, this is a pre-condition of the fix-point algorithm. Method n can now be modeled exactly like method m, i.e., by ignoring all sub-blocks containing recursive method calls during the modeling process. The above procedure is repeated until initial FDMs of all methods of a SCC are available.

Example 13.4.1 *Let us now have a look at the computation of the initial models of the methods testPosition(), advance(), and next() of class ConcreteQueen, which as depicted in Figure 13.6 all lie in the same SCC, i.e., are recursive with each other.*

Method testPosition() *is the root node within the SCC, because it is the only method called from outside the SCC, i.e., by method first() (see Figure 13.6). Further on, method testPosition() does not contain any top-level recursive method calls, because its only recursive method call, i.e., the one to method advance(), is nested in the body of the* ***while*** *loop in line 2 (see Figure 13.3). Hence,* $isRecursiveAtTopLevel(testPosition()) = false$ *holds. As described above we only model the method's top-level block and leave out the body of statement line 2, which contains a recursive method call. The resulting model in the initialization iteration, i.e., iteration 0, reads as follows. Note that all methods are denoted by their simple names.*

Recursive DFDM$^0_{testPosition()}$

$St.1: \; success_1 \leftarrow \{\{true\}, \{\}, \{\}, \{\}\}$

$St.3: \; \_result_1 \leftarrow \{\{\}, \{success_1\}, \{\}, \{\}\}$

The following locations are created:

0: *the receiver of method testPosition()*

1: *default location for 0.neighbor*

2: *default location 0.neighbor imported from method checkRow(int r, int c)*

Method advance() *can also be handled in step I of Algorithm 13.4.1, because its only recursive method call, i.e., the one to method next(), is nested in the then-branch of a selection statement. Hence,* $isRecursiveAtTopLevel(advance()) = false$ *holds. By ignoring the then-block we get the following model in iteration 0:*

Recursive DFDM$^0_{advance()}$

$St.1: \; success_1 \leftarrow \{\{true\}, \{\}, \{\}, \{\}\}$

$St.2: \; 0.row_1 \leftarrow \{\{1, 8\}, \{0.row_0\}, \{\}, \{\}\}$

$St.3: \; \_result_1 \leftarrow \{\{\}, \{success_1\}, \{\}, \{\}\}$

Note that for method advance() only default location 0 is created.

Method next() *does contain a top-level recursive method call, i.e., the call to method advance(). Consequently,* $isRecursiveAtTopLevel(next()) = true$ *holds. Therefore, we cannot model next() in step I of Algorithm 13.4.1, but have to rely on the initial models of advance() and testPosition(). By modeling all statements and blocks of next() we get the following model in iteration 0:*

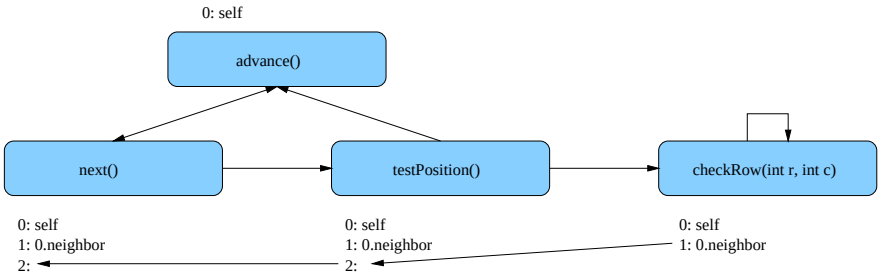


Figure 13.7: Locations created in iteration 0

```
      int rec1(int x, int y) {
1.      if (x == 0) {
1.1.      c = x;
1.2.      return 0; }
        else {
1.3.      c = x+y;
1.4.      return rec1(x-1,y); }
      }
```

Figure 13.8: Example method *rec1(int x)*

Recursive DFDM$^{0}_{next()}$

$St.1: \ 0.row_1 \leftarrow \{\{1,8\},\{0.row_0\},\{advance()\},\{\}\}$

$St.1: \ success_1 \leftarrow \{\{true\},\{\},\{advance()\},\{\}\}$

$St.2: \ success_2 \leftarrow \{\{true\},\{success_1\},\{testPosition()\},\{\}\}$

$St.3: \ \_result_1 \leftarrow \{\{\},\{success_2\},\{\},\{\}\}$

The following locations are created:

0: *the receiver of method next()*

1: *default location 0.neighbor imported from method testPosition()*

2: *default location 0.neighbor imported from method checkRow(int r, int c) via method testPosition()*

The locations created for all three methods in iteration 0 of the fix-point algorithm are depicted in Figure 13.7.

The use of the predicate $isRecursiveAtTopLevel(m)$ during the initialization phase of the fix-point algorithm (see Algorithm 13.4.1) seems rather complicated and one could argue that we could compute all initial models by simply leaving out all recursive method calls instead of whole blocks. This, however, is not

possible in the general case. When recursive method calls are left out during iteration 0 of the fix-point algorithm, incorrect FDs can be computed, which are never eliminated in later iterations. The following example is designed to demonstrate this problem.

Example 13.4.2 *Consider method rec1(int x, int y), which is depicted in Figure 13.8. Assume that variable c is defined as a static variable of the same class defining rec1(int x, int y). Obviously, variable c depends on parameter x, which appears in the condition of the* ***if*** *statement in line 1 and on the right-hand sides of assignments to c in both branches of statement 1. However, c is not influenced by parameter y, although y appears on the right-hand side of the variable assignment in line 1.3. This is, because the assignment in line 1.1 is always executed as the last assignment before rec1(int x, int y) terminates. If we use Algorithm 13.4.1 to compute an initial model of method rec1(int x, int y), we only model the then-branch of statement 1 and get the following FD:*

$St.1: \; c \leftarrow \{\{0\}, \{x_0\}, \{\}, \{\}\}$

If we decide to model the else-block as well and simply ignore the recursive method call in line 1.4, we get the following FD, which contains an incorrect variable occurrence, i.e., y_0, on its right-hand side:

$St.1: \; c \leftarrow \{\{0\}, \{x_0, y_0\}, \{\}, \{\}\}$

13.4.2 Computing the next iteration

We now have to define how to compute the methods of a particular SCC in iteration i using the method models of the previous iteration $i-1$. Generally, this is done by modeling each method m of the SCC separately by applying the principles presented in previous sections. We always start with the root node of the SCC and then successively model all methods, which call other methods already modeled in iteration i. Note that generally this process is non-deterministic, since we do not define the exact sequence, in which these methods are modeled. Since we can now rely on the fact that all other methods in the SCC have already been modeled in iteration $i-1$ at the latest, all statements of m can be modeled, even if they contain recursive method calls. This is also true for selection and loop statements, whose sub-blocks can now fully be modeled. The exact procedure of computing the *DFDM* of method m in iteration i, i.e., $DFDM_m^i$, is given by Algorithm 13.4.2:

Algorithm 13.4.2

Direct recursion: If in method m a recursive call to m is found, we always have to use m's model from iteration $i-1$, i.e., $sum(DFDM_m^{i-1})$, to model the recursive method call.

Indirect recursion: If in method m a recursive call to another method n in the same SCC appears, the most recent model of n is used. This can either

be the model from iteration i, i.e., $DFDM_n^i$, if n is modeled before m in iteration i, or the model from iteration $i-1$, i.e., $DFDM_n^{i-1}$, otherwise. Note that always using the most recent model leads to a faster convergence of the fix-point algorithm.

Handling variable dependencies: When modeling recursive methods, which do not create any locations (neither default nor new locations) this approach eventually leads to a stable model for all methods of the SCC. Obviously, FDs computed in previous iterations for the model of a method m can never be invalidated in the current iteration i, which leads to a monotonically increasing amount of FDs for method m over the course of the fix-point algorithm. More formally, we define the following lemma:

Lemma 13.4.1 *Let FDM_b be the FDM and $sum(FDM_b)$ the summarized FDM of a block b. Let further FDM_b' be a modification of FDM_b, which results from adding variable v to one of the FDs' right-hand sides. Then all FDs in $sum(FDM_b')$ are greater or equal than their counterparts in $sum(FDM_b)$. We write $sum(FDM_b') \geq sum(FDM_b)$. This also means that through the insertion of v no dependency from $sum(FDM_b)$ can be lost.*

Proof: Let us remember the way $sum(FDM_b)$ is computed. This is done by starting with the last FD for each variable v in b and successively substituting the VOs on the FD's right-hand side with a prior FD's right-hand side, whose left-hand side matches the VO on the FD's right-hand side (see Section 6.6). In case of multiple variables on a right-hand side, all variables have to be substituted. If we now add a variable w to any of the FD's right-hand sides, more substitutions will take place during the summarizing algorithm, but all substitutions performed previously will still be performed. This leads to an increased $sum(FDM_b)$. **Q.E.D.**

Theorem 13.4.2 *The number of FDs computed for a given method m, i.e., the number of variables changing their values during the execution of m, is constant after the first iteration. The sizes of all FDs computed for method m is monotonically increasing during the whole fix-point algorithm. We write $DFDM_m^i \geq DFDM_m^{i-1} \; \forall \, i \geq 1$. This means that if a variable v depends on another variable w in iteration i, v also depends on w in iteration $i+1$.*

Proof: We prove the above theorem by induction over the number of iterations. Let us first assume that method m is directly recursive.

Iteration 0: During the initialization iteration of the fix-point algorithm all FDs arising from blocks without recursive method calls are created. If no infinite

recursion is detected (see Section 13.3), we get a model for at least the top-level block of m. Note that there exists at least one sub-block, which has not been modeled completely.

Iteration 1: In the first iteration all blocks are considered. The ones already modeled in iteration 0 are, again, considered, which leads to the same FDs as in iteration 0. All other sub-blocks, i.e., all blocks containing recursive method calls, are modeled for the first time. The modeling of these sub-blocks results in the creation of new FDs for these sub-blocks. However, additional FDs stemming from a sub-block of statement s can never lead to the deletion of FDs at the top-level of s. In case of selection statements the modeling of an additional branch always leads to the same or more FDs at the selection statement's top-level, because the top-level FDs are computed by unifying all branches (see Section 10.3). The situation is similar for loop statements. If the loop body gets modeled, this can never invalidate the top-level FDs already computed for the loop condition (see Section 10.4). Note that now FDs for all variables changed in m are created.

Iteration i: We assume that $DFDM_m^i \geq DFDM_m^{i-1}$ holds.

Iteration i + 1: Let mc be a recursive method call in block b of method m. When modeling mc in iteration i, we make use of the summarized model of m in iteration $i-1$, i.e., $sum(DFDM_m^{i-1})$. Obviously, all imported FDs stem from variables, for which an FD already exists. Thus, the number of FDs does not increase any more. Since $DFDM_m^i \geq DFDM_m^{i-1}$ holds, there might exist larger FDs in $DFDM_m^i$ than in $DFDM_m^{i-1}$, which are now imported into $DFDM_m^{i+1}$ in iteration $i+1$. According to Lemma 13.4.1 this can only result in larger FDs but never lead to the deletion of existing ones. Thus the number of FDs stays constant, but their sizes are monotonically increasing.

Q.E.D.

These results can easily be applied to the general case of indirect recursions, where it cannot always be determined a priori, which model of the called method n (the one from iteration i or the one from iteration $i-1$) is used in iteration i. However, all blocks of the calling method m are modeled in iteration 1 at the latest and the modeling of additional blocks never invalidates existing FDs (as above). The FDs imported from method n are in either case greater than or equal to the ones imported in iteration $i-1$, which using Lemma 13.4.1 leads to a monotonically increasing size of all FDs of method m.

It is clear that after a finite amount of iterations no new FDs are introduced into the model of method m and thus a fix-point is reached. The algorithm terminates, when the models of all methods of the currently analyzed SCC in iteration i equal their respective counterparts in iteration $i-1$ (see Section 13.4.3).

Handling locations: Now consider a recursive method, which creates new objects, i.e., locations, in its body. Method *rec2(int t)*, which is depicted in

```
      C rec2(int t) {
        C x;
1.      if (t > 0)
1.1.      x = rec2(t-1);
        else
1.2.      x = new C();
2.      return x;
      }
```

Figure 13.9: Example method *rec2(int t)*

Figure 13.9 is directly recursive and therefore constitutes a SCC on its own. In statement line 1.2 it contains a class instance creation expression, which at run-time creates a new instance of class C if the condition $t > 0$ evaluates to `true`.

During the initialization iteration ($i = 0$), the then-branch of statement 1 is ignored, because it contains a recursive method call in statement line 1.1. The initial model of method *rec2(int t)*, $DFDM^0_{\text{rec2(int t)}}$, therefore only contains one FD arising from statement 1.2. Two locations are created in iteration 0, location 0 to represent the receiver of method *rec2(int t)* (i.e., **this**) and location 1 for the new instance of class C created in line 1.2. We write $loc(DFDM^0_{\text{rec2(int t)}}) = \{0, 1\}$. In the first iteration all statements are modeled building on the initial model $DFDM^0_{\text{rec2(int t)}}$. We get two FDs (for statements 1.1 and 1.2) and three locations, location 0, location 1 imported from $DFDM^0_{\text{rec2(int t)}}$ via the recursive method call in statement 1.1, and a new location 3 representing the newly created instance of class C in statement 1.2. Thus, $loc(DFDM^1_{\text{rec2(int t)}}) = \{0, 1, 2\}$. If we continue the modeling process we get $loc(DEFM^2_{\text{rec2(int t)}}) = \{0, 1, 2, 3\}$ in iteration 2 (location 0, locations 1 and 2 imported from $DFDM^0_{\text{rec2(int t)}}$ and $DFDM^1_{\text{rec2(int t)}}$, respectively, and a new location 4), $loc(DFDM^3_{\text{rec2(int t)}}) = \{0, 1, 2, 3, 4\}$ in iteration 3 etc...

Obviously, the explosion of locations during the modeling of recursive methods prevents the algorithm from reaching a fix-point. With an increasing amount of locations, we also encounter an infinite amount of changes in the FDs arising from the recursive method call. Therefore, the exit condition requiring all FDs to stay stable is never satisfied leading to the non-termination of the modeling algorithm.

This behavior is due to the fact that we are still performing a purely static analysis of method *rec2(int t)*. We therefore do not know how often method *rec2(int t)* gets called at run-time and how many instances of class C are created during program execution. This problem very much resembles the problems we are facing when computing the *DFDM* of a loop statement with the creation of locations in the loop body. We therefore give a similar solution by introducing multiple-locations for recursive methods. Such a concept changes the semantics of the model, but guarantees that a fix-point can be found in a finite number of

iterations.

Consider a location l_1, which is either created in a recursive method m or imported into m through a call to some other method n, which is not part of the same SCC as m, i.e., m is not recursive with n. All statements of m, which are modeled in iteration i, are also modeled in iteration $i+1$, leading to a monotonically increasing amount of FDs computed for m. This means that in iteration $i+1$ another location l_2 is created by exactly the same statement or method call as in iteration i. The two locations, l_1 and l_2, are two different instances of the same class, i.e., two different locations of the same type, but are known to have arisen from the same class instance creation expression. In iteration $i+1$ now both locations are visible. Whereas l_2 is created in iteration $i+1$ or imported through a call to method n, l_1 is imported through a recursive call of either m or another method o, which lies in the same SCC as m.

If we look at the keys of the locations l_1 and l_2, i.e., k_1 and k_2, we find, that both keys only differ in their method call path. This seems clear if we remember that both locations stem from the same source code position and were created for the same reference variable. The two method call paths differ in the way that k_1 includes a cycle at its front, which represents the sequence of recursive method calls in iteration i. In case of direct recursion, the cycle only consists of one element, i.e., the method call key of the recursive method itself. In case of indirect recursion, the cycle includes n elements with n being exactly the size of the modeled SCC.

In order to prevent the modeling algorithm from creating an infinite number of locations, we pool all locations arising from the same statement. The result of this approach is a multiple-location, which represents a finite but unknown number of locations created at run-time. Now the similarities with the modeling of loop statements become obvious. In our case one multiple-location stands for l_1, l_2, and all other locations arising from the same statement during the whole modeling process. As with loop statements, the resulting abstract multiple-location means a loss in precision of the resulting model, but makes sure that only a finite number of locations is created during the whole modeling process. See Section 10.4 for further properties of multiple-locations.

After all these explanations the algorithm for summarizing concrete locations to abstract multiple-locations seems straightforward. After each iteration of the modeling process all locations of the modeled method m have to be checked, whether they can be pooled to a multiple-location. This is the case, if the key of one location equals the key of another location after a complete method call cycle is removed in either of the two locations. More formally, we define:

Definition 13.4.1 *Let path be the method call path of an imported location l stored in its key k. We then define* $removeCycle(path) = path'$ *with*

- $path' = \{\langle mc_{i+1}, md_{i+1}\rangle, ...\langle mc_n, md_n\rangle\}$ *if* $path = \{\langle mc_1, md_1\rangle, ..., \langle mc_i, md_i\rangle, \langle mc_{i+1}, md_{i+1}\rangle, ..., \langle mc_n, md_n\rangle\}$ *and* $md_1 = md_{i+1}$.
- $removeCycle(path) = path$, *otherwise.*

Definition 13.4.2 *Let $k = \langle v, path \rangle$ be the key of an imported default location.* $removeCycle(k) = \langle v, removeCycle(path) \rangle$.

Definition 13.4.3 *Let $= \langle pos, path \rangle$ be the key of an imported new location.* $removeCycle(k) = \langle pos, removeCycle(path) \rangle$.

Two locations l_1 and l_2 with the keys k_1 and k_2 are summarized after iteration i iff $k_1 = removeCycle(k_2)$ or $k_2 = removeCycle(k_1)$. Applying this approach we can compute the model of a method m in SCC s in iteration i, i.e., $DFDM_m^i$, in the general case. We can now repeatedly compute the models of all methods within s until we eventually encounter the exit condition of the fix-point algorithm. The next section discusses the termination of the fix-point algorithm.

Example 13.4.3 *Let us come back to our example class ConcreteQueen and the SCC including the methods testPosition(), next(), and advance(). We can now compute iteration 1 by using the initial models of the three methods as described in Section 13.4.1 and all the models of all following iterations until a fix-point is eventually found.*

Let us first look at the locations, which are created in or imported into the individual methods in iteration 1. First, the methods testPosition() and next() are modeled, which both rely on the model of method advance from iteration 0. Therefore, no new locations are created in testPosition() and next(). During the modeling of advance() we for the first time create a default location for the variable 0.neighbor and import locations from next(), which leads to the locations depicted in Figure 13.10.

In iteration 2 location 3 of method advance(), which originally was imported from testPosition() via next(), is re-imported into method testPosition() and then into next(). Figure 13.11 shows the locations in iteration 2 with the bold line highlighting the re-importation process. However, in iteration 2 of the fix-point algorithm the following interesting things happen:

- *Clearly, after iteration 1 all locations have been created in their original methods. This is obvious, if we remember that in iteration 1 at the latest all statements and sub-blocks of all methods are modeled. However, only in iteration 2 all locations are present in all methods, because it takes one extra cycle to import a certain location to any other method within the same SCC.*

- *In iteration 2 for the first time a location is re-imported into method testPosition() and next(). These locations stem from method checkRow(int r, int c) and are first brought to the methods testPosition() and next() in the initialization iteration. In iteration 2 they are re-imported via method advance() and thus pooled to a single multiple-location during the summarizing process of testPosition() and next(), respectively.*

- *Due to the pooling of the re-imported locations, the number of locations after iteration 1 and iteration 2 stays constant. This will be the case for all following iterations, because any re-imported locations are immediately contracted to a multiple-location.*

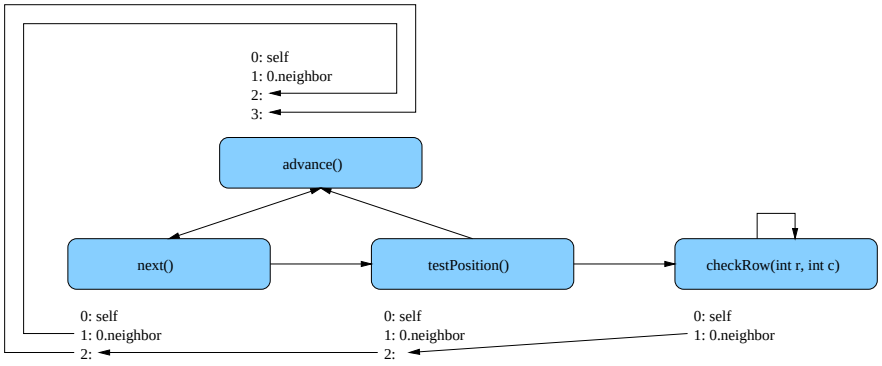


Figure 13.10: Locations created in iteration 1

Let us now have a look at the models of method testPosition() in iteration 1 and iteration 2. Due to the increasing number of locations the numbers and sizes of FDs increase, too. The following models show (for readability purposes) the variable dependencies arising from method testPosition() in iteration 1 and iteration 2, respectively:

Recursive DFDM$^{1}_{testPosition()}$

$St.1: \; success_1 \leftarrow \{\}$

$St.2: \; 0.row_1 \leftarrow \{0.col_0, 1.col_0, 2.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0, success_1, 0.row_0, 1.row_0, 2.row_0\}$

$St.2: \; success_2 \leftarrow \{0.col_0, 1.col_0, 2.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0, success_1, 0.row_0, 1.row_0, 2.row_0\}$

$St.3: \; \_result_1 \leftarrow \{success_2\}$

Recursive DFDM$^{2}_{testPosition()}$

$St.1: \; success_1 \leftarrow \{\}$

$St.2: \; 0.row_1 \leftarrow \{success_1, 0.row_0, 1.row_0, 2.row_0, 3.row_0, 0.col_0, 1.col_0, 2.col_0, 3.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0, 3.neighbor_0\}$

$St.2: \; 1.row_1 \leftarrow \{success_1, 0.row_0, 1.row_0, 2.row_0, 3.row_0, 0.col_0, 1.col_0, 2.col_0, 3.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0, 3.neighbor_0\}$

$St.2: \; success_2 \leftarrow \{success_1, 0.row_0, 1.row_0, 2.row_0, 3.row_0, 0.col_0, 1.col_0, 2.col_0, 3.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0, 3.neighbor_0\}$

$St.3: \; \_result_1 \leftarrow \{success_2\}$

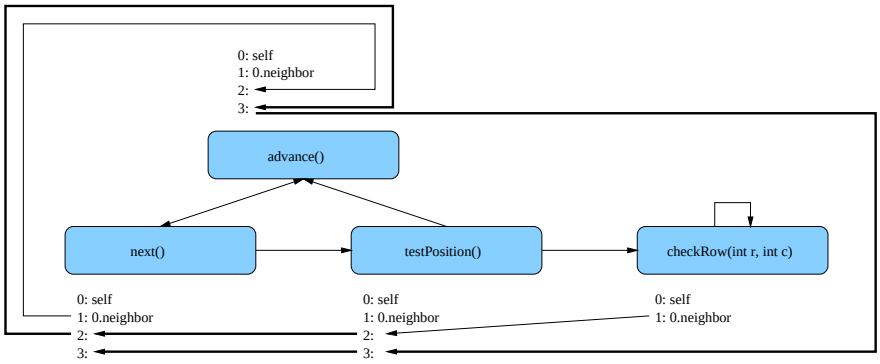


Figure 13.11: Locations created in iteration 2

13.4.3 Termination of the fix-point algorithm

The last step is to define the exit condition of the fix-point algorithm. As already mentioned in previous sections, the fix-point is reached, when the models of all methods in a given SCC in iteration i equal their respective counterparts in iteration $i-1$. As we saw the numbers and sizes of the FDs computed for a particular method are monotonically increasing during the fix-point algorithm. Hence, the fix-point is reached once no new FDs in iteration i emerge and all FDs remain unchanged in comparison to iteration $i-1$. In this section we show that a fix-point is always reached after a finite number of iterations.

Definition 13.4.4 *A fix-point of the FD modeling algorithm for a given SCC, s, is reached in iteration i iff $DFDM_m^i = DFDM_m^{i-1}\ \forall m \in s$.*

> ***JADE:*** *The* `JADE` *system indexes its locations by their position in the source code. The same location l can be indexed differently in two consecutive iterations, if additional locations are created or imported into the method before the creation of l. Therefore, before the exit condition can be checked, all locations of one of the two FDMs have to be set to indices as they appear in the other FDM. The renaming function is straightforward and is thus not described in this work.*

The proposed algorithm is only of value if we can prove that in all cases a fix-point is reached after a finite number of iterations. Before we show this property of the fix-point algorithm, we give the following theorems:

Theorem 13.4.3 *The total number of locations created for method m does not exceed the sum of all explicitly created and default locations of all methods in the same SCC as m.*

Proof: Each location l can either be created in m directly or, if it is created in another method n called by m, be imported into m. First, consider all default

and new locations of method m. All these locations are created in the initialization iteration or the following iteration for the first time depending on recursive method calls in m. In the following iterations exactly the same locations are created. The amount of explicitly created locations can therefore never increase, except for the initialization and the first iteration. Second, assume that location l is imported into m through a method call. We distinguish two cases:

- l originates from a statement s of method m, i.e., m will find its own signature at the beginning of the method call path of l. In this case l is immediately pooled with the location, which is created for statement s in the current iteration. The overall amount of locations stays the same.
- l originates from another method n. Given that the size of the SCC containing both, m and n, is finite, such a location is eventually re-imported into method n after a finite number of iterations, where it will be pooled as described in case 1. The number of locations imported from other methods is therefore finite.

Therefore, the total number of locations created for method m is always finite and equal to the specified number. **Q.E.D.**

Theorem 13.4.4 *Consider a SCC with the root node m. Let n be an arbitrary node in the SCC. Let p be the shortest method call path leading from m to n. Then, the maximum number of iterations needed to import a location l created in n into the root node m is given by $|p| + 1$.*

Proof: We proof Theorem 13.4.4 by induction over the length of p, i.e., $|p|$.

$|p| = 0$**:** This is the case, if $m = n$. Obviously, all locations are created in iteration 1 at the latest.

$|p| = 1$**:** This is the case, if n is directly called by m. All locations created in n in iteration 1 are imported into m in iteration 2.

$|p| = k$**:** We assume that all locations created in n are imported into m after iteration $k + 1$.

$|p| = k + 1$**:** If we now add a vertex to p, this means that an additional method call has to be performed in the method call sequence linking m to n. Therefore, we need one extra iteration to import all locations from n into m, i.e., $k + 2$ iterations.

Q.E.D.

Theorem 13.4.5 *Consider a SCC with the nodes $\{n_0, n_1, ...n_j\}$, where n_0 denotes the SCC's root node. Let $\{p_1, ..., p_j\}$ be the shortest paths from node n_i to the root node n_0 $\forall\, i \in \{1, ..., j\}$. Let further m be the length of the longest path in*

$\{p_1, ..., p_j\}$, i.e., $max(p_1, ..., p_j)$. Then, the number of iterations needed to create all locations for a given method m does not exceed $m + 1$.

Proof: From Theorem 13.4.4 it follows that in iteration $m + 1$ all locations created in all nodes of the SCC are imported into the root node n_0. Since the root node is always modeled first in each iterations, all other nodes import all locations from the root node in iteration $m + 1$, too. **Q.E.D.**

Based on Theorems 13.4.3 and 13.4.5 we can now prove that in all cases a fix-point is reached after a finite number of iterations, i.e., the following theorem holds:

Theorem 13.4.6 *The fix-point algorithm as described above always reaches a fix-point after a finite number of iterations.*

Proof: In Theorem 13.4.2 we have already shown that during the fix-point algorithm the sizes and numbers of FDs are monotonically increasing. This means that we can never encounter a situation, where two consecutive models of a given SCC oscillate around a fix-point, which is never reached. Either an infinite number of FDs or constituents are created or a fix-point is reached after a finite number of iterations.

Obviously, the number of variable occurrences, constants, and method declarations of a certain `Java` system is finite. If we do not consider locations, the worst case scenario is that everything (all constants, variables, methods) depends on everything, but the outcome still is a finite set of finite FDs. From Theorems 13.4.3 and 13.4.5 it follows that the total number of locations created for a given method is always finite. Therefore, only a finite number of (finite size) FDs is created during the course of the described fix-point algorithm. A fix-point is thus reached in all cases after a finite number of iterations. **Q.E.D.**

Example 13.4.4 *If we consider the SCC containing the methods testPosition(), next(), and advance(), we encounter a fix-point after iteration 4. This means that the summarized models of all three methods in iteration 4 equal the summarized models of the respective method in iteration 3. The following summarized model is computed for method testPosition() in iterations 3 and 4 of the fix-point algorithm. Note that the model contains only three locations, i.e., location 0, 1, and 2, after the locations 2 and 3 are contracted to a multiple-location 2.*

$\mathbf{sum}(\mathbf{FDM}_{testPosition()})$

$2.row_1 \leftarrow \{0.col_0, 1.col_0, 2.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0,$
$0.row_0, 1.row_0, 2.row_0\}$

$0.row_1 \leftarrow \{0.col_0, 1.col_0, 2.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0,$
$0.row_0, 1.row_0, 2.row_0\}$

$\_result_1 \leftarrow \{0.col_0, 1.col_0, 2.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0,$
$0.row_0, 1.row_0, 2.row_0\}$

$1.row_1 \leftarrow \{0.col_0, 1.col_0, 2.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0,$
$0.row_0, 1.row_0, 2.row_0\}$

$success_2 \leftarrow \{0.col_0, 1.col_0, 2.col_0, 0.neighbor_0, 1.neighbor_0, 2.neighbor_0,$
$0.row_0, 1.row_0, 2.row_0\}$

Chapter 14

The *SFDM*

In previous sections we introduce two types of FDMs, the *ETFDM* and the *DFDM*, and describe their construction in detail. In this chapter we propose a third FDM type, the Simplified Functional Dependency Model (*SFDM*). The *SFDM* is based on either the *ETFDM* or the *DFDM* and can automatically be derived from the two models. As we will see a higher level of abstraction leads to a new type of model, which is shorter and easier to read, but not as detailed and exact as the models proposed so far.

14.1 Basics of the *SFDM*

As discussed in Chapters 7 and 8, both models, the *ETFDM* and *DFDM* cover all FDs of a certain method by keeping locations and references separate. The advantage of this approach is that a broad range of program bugs can be located at statement level, whereas detailed information about all memory locations and variables exists. On the other hand, the *ETFDM* and *DFDM* are difficult to read and understand. The large amount of constants or run-time values, variables, and locations on the FDs' right-hand sides sometimes leads to slow debugging processes. Further on, a potential user has to have detailed knowledge about the underlying object structure when specifying an incorrect variable observation.

In contrast, the *SFDM* is easier to understand, includes only variables on the FDs' right-hand sides, and makes it easier for the user to specify observations. In order to define the *SFDM* we first divide the set of all variables into top-level variables and others using the following definition:

Definition 14.1.1 *A (reference) variable v is said to be a top-level (reference) variable iff it is one of the following:*

- *a local variable, i.e., of the form x*
- *a class variable, i.e., of the form $a.x$ with a being a simple or fully qualified class name*
- *an instance field of the currently modeled method, i.e., of the form $0.x$*

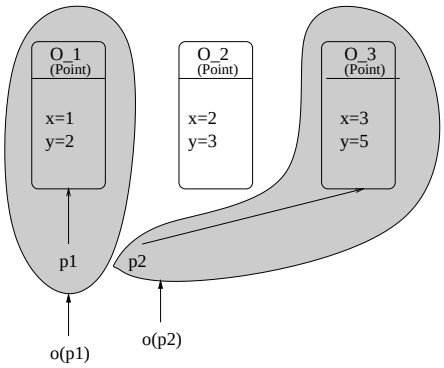


Figure 14.1: Abstract view of the Java system $system^5_{test()}$

Note that following this definition all instance fields of remote objects, i.e., all objects other than the receiver of the currently modeled method, are not top-level variables.

One of the key components of FDs in the *ETFDM* and *DFDM* is the variable occurrence, e.g., *vo* (see Section 6.1). Now we extend the concept of variable occurrences to the notion of object structures, which stand for a variable occurrence *vo* of variable v and all locations and variables currently (possibly) hidden behind v. In other words, an object structure contains the following elements: (1) A variable occurrence $vo = \langle v, b, i \rangle$ after which the object structure is named. (2) If v is of primitive type the value stored in v. (3) If v is of reference type the location l referenced by v and the object structures of all instance fields of l. More formally, we define:

Definition 14.1.2 *The complete object structure of a variable occurrence* $vo = \langle v, b, i \rangle$ *is defined as* $o(vo) = \{v \cup val(v)\}$ *if* v *is of primitive type and* $o(vo) = \{v \cup l \cup \{o(w) \mid w \in fields(l)\} \mid l = val(v)\}$ *if* v *is of reference type. Here* $val(v)$ *denotes the value of* v *if* v *is of primitive type and the location referenced by* v *otherwise.* $fields(l)$ *stands for the set of all instance fields of location* l*.*

We can now, in contrast to Chapter 5, specify the complete state of a Java system by considering the complete object structures of all top-level variable occurrences. In case of primitive variables this covers all these variables together with their current run-time values. In case of variables of reference type, not only the reference v itself, but also all locations and variables hidden behind v are part of the system description.

Example 14.1.1 *If we, again, look at method test() from Section 5.2 (see Figure 5.1), we get a new representation of the method's run-time state, which is depicted in Figure 14.1. After the execution of test() there are two top-level reference variable occurrences, p1 and p2. Both variables span a complete object structure, i.e.,* $o(p1)$ *and* $o(p2)$*, respectively. The state of the whole* Java *system can be specified by the set* $\{o(p1), o(p2)\}$*. Note that the third instance of*

class Point, which is not referenced by any (top-level) variable, is not part of any object structure and thus removed by the garbage collector.

As we will see in the following section, this new interpretation of a `Java` system's run-time state is the basis for the more abstract *SFDM*.

14.2 Simplified functional dependencies

Like in the *ETFDM* and *DFDM* the main component of the *SFDM* is a single FD. This FD now looks somewhat different from a FD of the models presented so far, because in the *SFDM* only variables are considered as possible influences of a FD's left-hand side. Moreover, instead of variable occurrences now object structures of top-level variable occurrences are used. The exact format of a simplified FD (SFD) is given as follows:

Definition 14.2.1 *A simplified functional dependency (SFD) of the variable occurrence* $vo = \langle v, b, i\rangle$ *is defined as the tuple* $SFD = \langle o(vo), DEP\rangle$*, i.e.,* SFD_{vo}*.*

Let us now, again, have a look at the meanings of the different parts of the SFD:

$\mathbf{o(vo) = o(\langle v, b, i\rangle)}$ is the SFD's left-hand side and stands for the complete object structure of vo. Variable v can be of primitive or of reference type. In case of v being of reference type it no longer simply denotes the reference, but all locations and references hidden behind the reference v as well. Here we see that it is no longer possible to distinguish between a reference and the location referenced by it. As we will see that makes the model much shorter and simpler, but also introduces a couple of interesting questions and problems. Now v can only be of the following forms:

- ***x*:** In this case x denotes a local variable of the currently modeled block or an instance variable of the class containing the currently modeled method. All other structures (object locations, instance fields of these objects, objects referenced by these instance fields, etc...) are hidden behind x and are so implicitly covered by the SFD.
- ***a.x*:** in this case x is a static variable and a denotes the fully qualified name of the class, in which x is defined. Again, if $a.x$ is of reference type, it stands for the reference and all locations and further references hidden behind $a.x$.

***DEP*:** is the SFD's right-hand side. In contrast to the *ETFDM* and *DFDM*, *DEP* only contains object structures of variable occurrences that (possibly) influence the FD's left-hand side. Therefore, *DEP* contains $o(vo)$, if $vo = \langle w, b, i\rangle \in VO$ stands for a variable occurrence influencing the FD's left-hand side. w can have the same form as v. It can thus be of primitive or of reference type. In case of w being of reference type it stands for the whole object structure behind the reference w.

14.3 Creating a *SFDM*

As mentioned above the *SFDM* can automatically be derived from either the *ETFDM* or the *DFDM*. This is done by successively simplifying each FD of the underlying FDM into one or more SFDs. The resulting *SFDM* can be defined as follows:

Definition 14.3.1 *The $SFDM_m$ for a given FDM_m is defined as $SFDM_m = \bigcup_{fd \in FDM_m} simplify(fd)$.*

The conversion from a FD, e.g., fd, to a set of SFDs is defined by function $S = simplify(fd)$, which can be computed using Algorithm 14.3.1:

Algorithm 14.3.1 Consider a FD $fd = \langle vo, DEP \rangle$, where $vo = \langle v, b, i \rangle$ and $DEP = \langle R, V, M, O \rangle$ or $DEP = \langle C, V, M, L \rangle$. Let further DEP' be the set of object structures of all top-level reference variables in V, i.e., $DEP' = \{o(vo) \mid vo \in V'\}$ where $V' \subseteq V$ is the subset of V containing all top-level variables occurrences. Then, compute the set of SFDs $S = simplify(fd)$ by distinguishing the following two cases:

Case 1: v is a top-level variable. Then, $S = \langle o(vo), DEP' \rangle$.

Case 2: v is not a top-level variable, i.e., it is of the form $n.x$ with $n > 0 \in L$ ($DFDM$) or $n > 0 \in O$ ($ETFDM$). Then, $S = \langle o(y), DEP'' \rangle \mid y \in Y\}$, where Y is defined as the set of all top-level variables, which at the given point within the program (possibly) reference n or another location n', which in turn contains references to n. DEP'' equals DEP' as defined above, only that sometimes self references have to be inserted, i.e., $DEP'' = DEP' \cup o(y)$. The introduction of self references is discussed in Section 14.4.

Finally, when computing a *SFDM* note the following points:

- The computation of Y can be done by a recursive algorithm, which in one step computes all references currently (possibly) pointing at location n. The resulting references are either top-level variables or include another location n' in their scope. In the latter case the reference variable has to be further resolved.

- During this process the indices of all variable occurrences are adapted in order to guarantee a correct sequence of occurrence indices from the method's beginning to its end.

- If multiple FDs for different variables in the same object structure arise from the same expression or statement, it is now possible that more than one SFD is computed for the same top-level variable v. In this case all SFDs have to be combined by forming the union of their right-hand sides.

- As a result, FDs for variable occurrences of reference type no longer simply denote that reference. Instead the dependency directly refers to the locations and references representing the local state of the referenced object (since these locations are no longer explicitly present in the model).
- A *SFDM* can be based on either an *ETFDM* or a *DFDM*. We should therefore speak about simplified *ETFDMs* (*SETFDMs*) and simplified *DFDMs* (*SDFDMs*). However, in this section we talk about *SFDMs* meaning FDMs based on either an *ETFDM* or a *DFDM*.
- All locations and instance fields with scopes other than 0 at an FD's right-hand side are simply deleted and not resolved to top-level references as the variable on the FD's left-hand side. There are two reasons for that: (1) In case of variables with a scope of $n > 0$ the top-level reference should already be part of V. This is, because a field access is modeled by considering not only the instance field, but also the reference defining the scope of the field access in the right-hand side of the arising FD. (2) As we will see locations are not needed in the final version of the model any more. They are important throughout the modeling process (i.e., during the computation of the *ETFDM* or *DFDM*) in order to solve aliasing problems. Section 14.4 deals with aliasing problems in *SFDMs*.
- As described in Algorithm 14.3.1 in some cases self dependencies are added to SFDs. The reason for this approach and problems arising in its context are discussed in Section 14.4.

Example 14.3.1 *Let us come back to our example of Section 5.2, which is depicted in Figure 5.1. If we apply Algorithm 14.3.1 to either the ETFDM or the DFDM of method test() (see Chapters 7 and 8) we get the following SFDM:*

$\mathbf{SFDM}_{test()}$

$St.1: \; p1\_1 \leftarrow \{\}$

$St.2: \; p2\_1 \leftarrow \{\}$

$St.3: \; p1\_2 \leftarrow \{p1\_1\}$

$St.4: \; p1\_3 \leftarrow \{p1\_2\}$

$St.5: \; p2\_2 \leftarrow \{p1\_3, p2\_1\}$

14.4 Handling aliasing with the *SFDM*

Both model types, the *DFDM* and *ETFDM*, distinguish between references, i.e., variables of reference type, and object locations. As discussed in Section 7.5 by doing this we implicitly solve all potential aliasing problems occurring in the `Java` source code. The reason for that is quite obvious, because references to locations and the locations themselves are modeled by different model constructs.

The *SFDM* introduces a new notion of aliasing. We no longer speak about aliasing only if two variables of reference type reference the same object, but also if two variables of reference type reference two different object structures, which in turn hold references to the same object. Due to the changed model semantics of the *SFDM* we redefine the aliasing problem as follows:

Definition 14.4.1 *Two variables of reference type, v and w, are said to cause aliasing, if there exists a location l, which is part of both object structures, the one referenced by v and the other modeled by w. During the modeling process we face an aliasing problem, if the content of l is altered by accessing l through one of the two variables, say v. A modification of l should then be visible in the model not only for an access of l through v, but also for an access of l through w.*

In contrast to the *ETFDM* and *DFDM*, references and locations are no longer kept separate. Moreover, it is the key property of the *SFDM* that these model components are contracted into a single object structure denoted by the name of the top-level reference variable. As a direct consequence, aliasing problems are no longer explicitly covered. In order to make the *SFDM* suitable to handle aliasing problems during the transformation of a FD to a SFD we have to ensure the following two points:

Reference resolution: The left-hand side of a FD, i.e., vo, is resolved by computing *all* top-level references currently pointing at the object structure o including vo. By this we make sure that if the contents of o is changed, not only a single SFD is created for variable vo, but one SFD for each top-level reference. Hence, we assure that model components, i.e., all top-level references, stay up to date.

Self dependencies: Due to the changed semantics of the *SFDM* we are facing another problem, which can be described as some sort of frame problem: If a variable w, which is part of an object structure o hidden behind variable v changes its value, the rest of the object structure stays unchanged. This fact has to be considered, when computing a SFD arising from the change of variable w. If we compute a SFD for variable v with only such constructs on its right-hand side, which cause the change of w, we delete all existing dependencies between v and the rest of o. A very simple and efficient solution is to introduce self dependencies, i.e., assure that v is always part of the SFD's right-hand side. In most cases this postulation is met anyway, but in other cases self dependencies have to be created explicitly. Note that self dependencies have only to be introduced if not the whole object structure gets modified. This is the case, if the top-level reference itself is changed. In other words, if the *ETFDM* or the *DFDM* of a given expression or statement contains a FD for a top-level reference variable v, then the SFD for variable v does not have to contain a self dependency.

Example 14.4.1 *Let us now consider the example from Section 5.2 (see Figure 5.1) and method aliasing(), which is depicted in Figure 14.2. In lines 1 and*

```
      void aliasing() {
        Point p1, p2;
1.      p1 = new Point(0,0);
2.      p2 = p1;
3.      p1.doubleXValue();
4.      int tmp = p2.x;
      }
```

Figure 14.2: Example method *aliasing()*

2 we find two variables of reference type, p1 and p2, pointing at the instance of class Point created in line 1. In the context of the SFDM these two variables are top-level references comprising the reference itself and the point object. If we now change the content of the point object in line 3, both top-level references are influenced. This gets obvious by looking at the output of Algorithm 14.3.1, which reads as follows:

$\mathbf{SFDM}_{aliasing()}$

$St.1: \; p1_1 \leftarrow \{\}$

$St.2: \; p2_1 \leftarrow \{p1_1\}$

$St.3: \; p1_2 \leftarrow \{p1_1\}$

$St.3: \; p2_2 \leftarrow \{p2_1, p1_1\}$

$St.4: \; tmp_1 \leftarrow \{p2_2\}$

If we now assume the object structure p1 to be incorrect after the execution of statement line 4, we find statements 1 and 3 as possible culprits. This seems to be clear as only these two statements change either variable p1 or the point object. If on the other hand we assume the object structure p2 to be incorrect, we not only get statement 1 and 2 as possible diagnoses, but also statement line 3, where the object structure (not the reference) p2 is modified. It should now become obvious how the SFDM handles aliasing problems.

Note that in the second SFD for line 3 a self dependency is inserted. Without this dependency we get line 3 as a possible diagnosis and thus solve the aliasing problem, but lose all information about dependencies of the rest of the object space p2. By adding the self dependency we make sure that statements 1 and 2 can also be reached from the SFD for statement line 3.

14.5 Properties of the *SFDM*

Similar to the *ETFDM* and *DFDM*, we shortly summarize the benefits and drawbacks of the *SFDM*. The quality of various *SFDMs* for debugging has been tested using different test-cases. The exact results are presented in Part III of this work. The most important properties of the *SFDM* are:

- As already discussed the *SFDM* is based on either the *ETFDM* or the *DFDM*. Therefore, the main properties of a *SFDM* depend on the general properties of the underlying model type and are different for *SFDMs* based on *ETFDMs* and *SFDMs* based on *DFDMs*.

- Due to the changed model semantics *SFDMs* are only complete in relation to the new interpretation of object structures. Note that they are no longer complete as far as individual variables and locations are regarded.

- As mentioned above the *SFDM* introduces a new variant of the aliasing problem. As shown this problem can be solved by repeatedly resolving references to given locations and by adding self dependencies in certain cases.

Advantages of the *SFDM*:

- The *SFDM* is easier to read and understand, mainly because it does not contain locations and variables with scopes.

- As a consequence, the *SFDM* contains less FDs with fewer variables, which, as we will see, leads to a faster diagnosis process, i.e., diagnoses are computed quicker.

- With the *SFDM*, observations might be easier to specify for the user, who generally thinks in terms of references and not in terms of object locations

Disadvantages of the *SFDM*:

- Due to the introduction of object structures the system can no longer distinguish between references to objects and object locations, which in turn leads to a loss of information. As a consequence, the granularity of the model becomes coarser.

- Observations cannot be specified as precisely as with either the *ETFDM* or the *DFDM*. A user, who knows the exact location of an observed error, has to specify a reference pointing at that location rather than the location itself.

- The loss of information has to be paid off with larger diagnosis sets. This, as we will see in Part III, will lead to an increased amount of variable queries put to the user and to a slower convergence to the correct diagnosis.

The introduction of self dependencies is discussed in Section 14.4. Unfortunately, this concept does not only allow for an efficient handling of aliasing, but also poses a couple of new problems, which are shortly highlighted on the basis of the following example.

```
     void sfdm1() {
1.     obj o = new obj();
2.     o.set(3);
3.     o.set(1);
4.     o.set(5);
5.     o.set(2);
6.     o.set(4);
     }
```

Figure 14.3: Example method *sfdm1()*

Example 14.5.1 *Consider the example method sfdm1(), which is depicted in Figure 14.3. In statement line 1 it creates an instance of class obj, whose instance field field is left uninitialized (and thus stores its default value). In the following statements, i.e., lines 2 to 6, the value of field is then repeatedly set to a new value, which each time overrides the value previously held by field. When we construct the DFDM of sfdm1() we get the following model. Assume that we specify the value of variable field, i.e., 1.field, to be incorrect, we get two possible diagnoses, i.e., statement 1 or statement 6 to contain the bug.*

$\mathbf{DFDM}_{sfdm1()}$

$St.1: \ 1.field\_1 \leftarrow \{\{\},\{\},\{obj()\},\{\}\}$

$St.1: \ o\_1 \leftarrow \{\{\},\{\},\{obj()\},\{1\}\}$

$St.2: \ 1.field\_2 \leftarrow \{\{3\},\{o\_1\},\{set(int)\},\{\}\}$

$St.3: \ 1.field\_3 \leftarrow \{\{1\},\{o\_1\},\{set(int)\},\{\}\}$

$St.4: \ 1.field\_4 \leftarrow \{\{5\},\{o\_1\},\{set(int)\},\{\}\}$

$St.5: \ 1.field\_5 \leftarrow \{\{2\},\{o\_1\},\{set(int)\},\{\}\}$

$St.6: \ 1.field\_6 \leftarrow \{\{4\},\{o\_1\},\{set(int)\},\{\}\}$

If we now look at the corresponding SFDM of method sfdm1(), we find that each FD (apart from the first one) contains a self dependency. We can no longer specify variable 1.field to be incorrect, but only the object structure o to be in an incorrect state. This, however, results in all six statements being potential culprits, which highlights a major drawback of SFDMs as far as their ability to locate software faults is concerned.

SFDM$_{sfdm1()}$

$St.1: \; o\_1 \leftarrow \{\}$

$St.2: \; o\_2 \leftarrow \{o\_1\}$

$St.3: \; o\_3 \leftarrow \{o\_2\}$

$St.4: \; o\_4 \leftarrow \{o\_3\}$

$St.5: \; o\_5 \leftarrow \{o\_4\}$

$St.6: \; o\_6 \leftarrow \{o\_5\}$

Part III

Debugging

Chapter 15

System Descriptions

In Part II of this work we presented various ways of modeling `Java` systems. All described models cover static and dynamic properties of the respective `Java` systems and are thus suited to be used for fault localization. This part describes how the FDMs described in Part II can be used to debug a faulty `Java` method.

15.1 Diagnosis components

Diagnosis components are the constituents of every system description. They can directly be derived from the FDMs, whose computation is presented in detail in Chapter 12. The following types of diagnosis components are used for the debugging of `Java` systems at statement level:

Assignment components are the basic components directly derived from the FDM. These components store the statement, which they are associated with, together with a set of FDs, which arise from the modeling of this statement. The set of FDs always includes at least one FD, which models the data dependency introduced by the variable assignment itself. Further FDs might stem from side-effects, which are imported into the statement model through method calls inside the variable assignment.

Selection components are always associated with a selection statement and thus store an **if** or **switch** statement. The set of FDs stores all side-effect FDs arising from the selection statement's condition and all FDs, which belong to the summarized FDM of the statement's branches. In addition, every selection component stores all diagnosis components, which were computed for all branches of the statement. For example, a diagnosis component associated with an **if** statement contains two additional sets storing all diagnosis components of its then- and else-branch, respectively.

Loop components are always associated with a loop statement and thus store a **do**, **for**, or **while** statement. Similar to selection components, the FD set stores all side-effect FDs arising from the loop condition and all FDs belonging to the summarized FDM of the loop body. An additional set

of sets stores all diagnosis components of the loop body for all modeled iterations.

Note that in this work we focus on the debugging of `Java` methods at statement level. To be used for expression level debugging, the individual diagnosis components have to be slightly modified. Not all statements of a `Java` system have associated diagnosis components. If during the FD modeling process no FDs for a particular statement are computed, then no diagnosis component is created either. This means that during the debugging process these statements are not part of any diagnosis. The type of FDM computed during the modeling step (*ETFDM*, *DFDM*, or *SFDM*) has no influence on the resulting diagnosis components, only on the FDs stored in the individual components. However, there are some differences between *ETFDMs* and *SFDMs* based on an *ETFDM* on one hand and *DFDMs* and *SFDMs* based on a *DFDM* on the other:

- In combination with the *ETFDM* selection components always store the diagnosis components of only one branch of the statement, i.e., the branch being executed at run-time. The sets of diagnosis components for all other branches stay empty, because no FDs are created for these branches during the modeling process.
- When using the *DFDM*, only one model for a loop statement's body is created. This seems obvious, because the exact number of loop iterations is not known at compile-time. The set storing all sub-block models therefore always has only one element, i.e., a set of diagnosis components arising from the loop body. This, however, is not true for the *ETFDM*. When an evaluation trace is used during the modeling process, each loop iteration is modeled separately and results in a different FDM. Therefore, loop components, when using the *ETFDM*, contain i sets in their *subBlock* field with i being the number of iterations being performed at run-time.

15.2 System descriptions

A complete system description, as it is needed for the fault localization process, can now be automatically constructed from the existing diagnosis components. In this section we show how to construct a hierarchical system description, which consists of all top-level structures of its diagnosis components. Sub-structures stored in the individual components, e.g., for loop bodies or branches of selection statements, are not considered at this level. They, in turn, can be combined to system descriptions on their own, if needed. A complete system description consists of the following parts:

Components ($\mathcal{C}$) constituting a system description are the diagnosis components of a given `Java` method. The diagnosis components are automatically derived from the various FDMs. Their format is described in Section 15.2. A component $c \in \mathcal{C}$ can non-ambiguously be associated with a statement of the analyzed method.

Connections ($\mathcal{L}$) link a diagnosis component $c_1 \in \mathcal{C}$ to another diagnosis component $c_2 \in \mathcal{C}$ via the use of ports. They always lead from a component's output port to another component's input port. Note that a connection $l \in \mathcal{L}$ can be linked to multiple input ports, but only one output port. As we will see there exists a one-to-one correspondence between connections and variable occurrences, i.e., each connection in a diagnosis system can non-ambiguously be associated with exactly one variable occurrence.

Ports ($\mathcal{P}$) are used to mount connections to diagnosis components. An individual port $p \in \mathcal{P}$ can non-ambiguously be associated with a variable occurrence vo of the analyzed `Java` method. We write $vo(p)$. Ports can be divided into two disjoint sets, input ports ($\mathcal{I}$) and output ports ($\mathcal{O}$). An input port $i \in \mathcal{I}$ of a diagnosis component $c \in \mathcal{C}$ stands for a variable occurrence influencing the outcome of component c. An output component $o \in \mathcal{O}$ of component c, on the other hand, models a variable occurrence, whose value is altered in the statement associated with component c.

> ***JADE:*** *The* ***JADE*** *system currently only uses variables as influencing factors of diagnosis components. Run-time values, constants and method calls are not used in the system description. For the sake of simplicity in this work only variables are used, too.*

As mentioned above the diagnosis components of our system descriptions can directly be derived from the FDMs. In order to get a complete system description we have to connect the existing components via connections and ports. The algorithm works as follows:

Algorithm 15.2.1

- $\forall c \in \mathcal{C}$: create an input port $i \in \mathcal{I} \forall\, v \in V_i$, where V_i stands for all variables appearing on the right-hand side of one of the FDs of c.
- $\forall c \in \mathcal{C}$: create an output port $o \in \mathcal{O}\ \forall\, v \in V_o$, where V_o stands for all variables appearing on the left-hand side of one of the FDs of c.
- $\forall o \in \mathcal{O}$ create a connection $l \in \mathcal{L}$ leading from o to all input ports $\{i \mid i \in \mathcal{I} \wedge vo(o) = vo(i)\}$. If no such input port i exists, a connection l is created and called an *output connection* of the diagnosis system.
- $\forall i \in \mathcal{I}$, which are not assigned to any connection, create a connection $l \in \mathcal{L}$. This connection is called an *input connection* of the system.

Example 15.2.1 *Consider method test() of class Point, which is depicted in Figure 5.1.* $DFDM_{test()}$ *is given in Section 12.1. We can now collect all FDs of the model in five diagnosis components representing the five statements of*

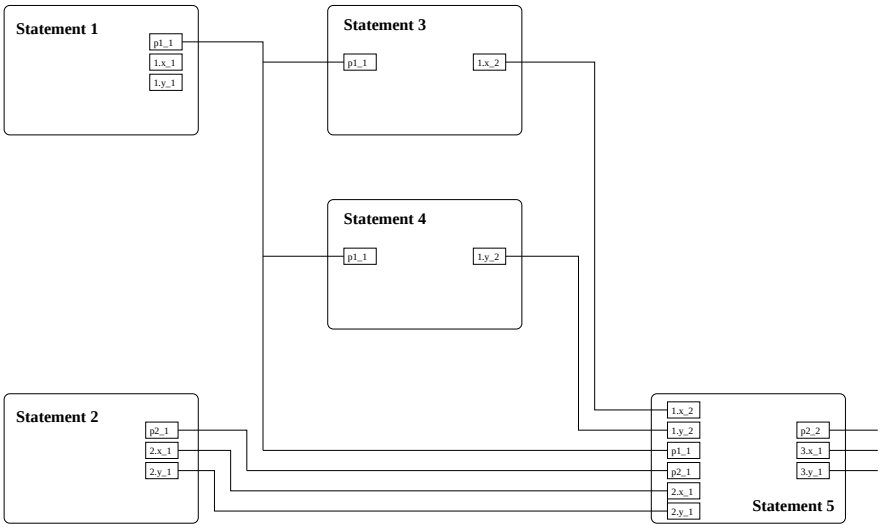


Figure 15.1: System description of method *test()*

test(). If we use these diagnosis components as input for the algorithm presented above, we get a complete system description of method test(), which is depicted in Figure 15.1.

Note that a system description as depicted in Figure 15.1 shows the components and structure of a given diagnosis problem. The exact behavior of the individual components is only defined in connection with the set of FDs stored in each component, which internally connects a component's input and output ports.

Chapter 16

Computing Diagnoses

As discussed in Chapter 3 a diagnosis system is defined as the pair $(SD, COMP)$, where SD is a logical model describing the correct behavior of the system and $COMP$ a set of components. Whereas the set of components can directly be taken from the used model, the system description as defined in Chapter 15 has to be transformed into a logical representation, i.e., SD, taking into account the structure of the system and the behavior of the individual components. This chapter describes how a logical model can be constructed and then in combination with concrete observations be used to compute diagnoses.

16.1 Logical model description

In order to generate a logical representation, i.e., a model, of a given method m we successively convert all FDs of the underlying FDM_m into logical sentences defining part of the behavior of the FD's component. The basic idea can be formulated as follows: *If a statement can be assumed to behave correctly and all variables used as input are also correct, then the statement's outputs must be correct, too.* Since FDs describe behavior only implicitly by describing influences between variable occurrences, we cannot make use of concrete variable values, but only speak about whether the value of a particular variable occurrence vo is correct (written as $ok(vo)$) or not (written $nok(vo)$). If we further use the predicate $\neg AB(C)$ ($AB(C)$) as defined in Chapter 3 to denote that component C, i.e., statement C, behaves correctly (incorrectly) and $fd(C)$ as the set of all FDs stored in component C, we get the following logical representation:

$$\forall_{\langle vo, DEP\rangle \in fd(C)} : \left[\neg AB(C) \wedge \bigwedge_{x \in DEP} ok(x) \rightarrow ok(vo)\right] \in SD$$

where $C \in COMP$ is a statement. In addition, we know that it is impossible that a variable value is known to be correct and incorrect at the same time. Therefore, we have to add the rule

$$ok(vo) \wedge nok(vo) \rightarrow \bot$$

to the model SD, for each variable occurrence vo in the program.

Example 16.1.1 *If we come back to method test() of class Point (see Figure 5.1), we can transform a FD method model of test(), i.e., $FDM_{test()}$, into a logical model using the algorithm described above. Using $DFDM_{test()}$ (see Section 12.1) we get the following logical sentences describing the behavior of the components $COMP = \{s1, s2, s3, s4, s5\}$:*

$$\neg AB(s1) \rightarrow ok(1.x_1)$$
$$\neg AB(s1) \rightarrow ok(1.y_1)$$
$$\neg AB(s1) \rightarrow ok(p1_1)$$
$$\neg AB(s2) \rightarrow ok(2.x_1)$$
$$\neg AB(s2) \rightarrow ok(2.y_1)$$
$$\neg AB(s2) \rightarrow ok(p2_1)$$
$$\neg AB(s3) \wedge ok(p1_1) \rightarrow ok(1.x_2)$$
$$\neg AB(s4) \wedge ok(p1_1) \rightarrow ok(1.y_2)$$
$$\neg AB(s5) \wedge ok(p1_1) \wedge ok(p2_1) \wedge ok(1.x_2) \wedge ok(2.x_1) \rightarrow ok(3.x_1)$$
$$\neg AB(s5) \wedge ok(p1_1) \wedge ok(p2_1) \wedge ok(1.y_2) \wedge ok(2.y_1) \rightarrow ok(3.y_1)$$
$$\neg AB(s5) \wedge ok(p1_1) \rightarrow ok(p2_2)$$

Finally, we complete the model by adding inconsistency sentences for all variable occurrences of the system, which reads as follows:

$ok(1.x_1) \wedge nok(1.x_1) \rightarrow \bot$, $ok(1.y_1) \wedge nok(1.y_1) \rightarrow \bot$, $ok(p1_1) \wedge nok(p1_1) \rightarrow \bot$,
$ok(2.x_1) \wedge nok(2.x_1) \rightarrow \bot$, $ok(2.y_1) \wedge nok(2.y_1) \rightarrow \bot$, $ok(p2_1) \wedge nok(p2_1) \rightarrow \bot$,
$ok(1.x_2) \wedge nok(1.x_2) \rightarrow \bot$, $ok(1.y_2) \wedge nok(1.y_2) \rightarrow \bot$, $ok(p2_2) \wedge nok(p2_2) \rightarrow \bot$
$ok(3.x_1) \wedge nok(3.x_1) \rightarrow \bot$, $ok(3.y_1) \wedge nok(3.y_2) \rightarrow \bot$,

16.2 Observations

Apart from the logical system description, the MBD approach requires a set of observations OBS in order to fully specify a diagnosis problem $(SD, COMP, OBS)$ (see Chapter 3). In the case of software debugging using FDMs, observations are given by logical sentences stating whether a particular variable occurrence $vo = \langle v, b, i \rangle$ is in a correct state, i.e., v has the correct value, or not. This can easily be determined by comparing the computed value of v with its expected value as defined by the intended behavior of the system. More formally, we write:

- $ok(vo)$, if the computed value of v matches its expected value and
- $nok(vo)$, otherwise

Example 16.2.1 *Assume that after the execution of method test() (see Figure 5.1) we expect the values of the instance fields x and y of the point object created in statement s5, i.e., the object at location 3 in the above model, to hold the values 4 and 5, respectively. We can then compare these values with the values computed by the* Java *run-time system, which are* $x = 3$ *and* $y = 5$*, and conclude the following: (1) Instance field x of location 3 is in an incorrect state and (2) instance field y of location 3 holds the correct value. Formally, we write* $nok(3.x_1) \wedge ok(3.y_1)$.

As discussed in Chapter 14, we cannot distinguish between a reference v being incorrect or the state of the object referenced by v being incorrect, when we are using *SFDMs* for debugging. Moreover, by specifying $nok(vo)$ for $vo = \langle v, b, i \rangle$ we specify something like *either variable v is referencing the wrong object or the object which is referenced by v is in an incorrect state.* Clearly, with the *SFDM* observations cannot be specified as concisely as with the *DFDM* or *ETFDM*, but on the other hand it seems easier for the user to determine the correctness of whole object structures than individual values and references.

16.3 Computing diagnoses

Since we have defined a complete diagnosis problem $(SD, COMP, OBS)$, we can use standard diagnosing algorithms taken from MBD together with a standard theorem prover in order to compute conflict sets and diagnoses of the used model (see Chapter 3). A diagnosing algorithm based on the computation of hitting sets of the collection of all conflict sets is proposed in [38]. This algorithm was improved by [17]. Conflict sets are computed using standard theorem provers.

Example 16.3.1 *If we compute diagnoses for the logical system description of method test() (see Figure 5.1) as specified in Sections 16.1 and 16.2, we get four single-diagnoses, i.e.,* $D = \{\{s1\}, \{s2\}, \{s3\}, \{s5\}\}$*. Informally, this can be explained as follows:*

- *Statement s5 obviously is a possible source of the bug, since it creates the object at location 3 through a call to method plus(Point p). Possible bugs within s5 are the call to an incorrect method, an incorrect receiver, or an incorrect argument.*

- *Statement s4 sets the instance fields y of location 1. Since this field is not involved in the computation of 3.x in statement s5, this statement is not a diagnosis*

- *Statement s3 alters the value of 1.x, which is used in statement 5 in the computation of the incorrect variable 3.x. Therefore, s3 has to be a diagnosis. Possible bugs within s3 are an incorrect constant on the assignment's right-hand side, an incorrect assignment operator, or an incorrect scope of the assignment's left-hand side.*

- *Statement s2 sets the value of 2.x, which is used in the computation of 3.x in statement s5. Therefore, s2 is a possible diagnosis. Potential bugs are incorrect arguments or the call to an incorrect constructor.*

- *Statement s1 is also a diagnosis, although this does not seem obvious. The reason is that p1 appears in the scope of the method call in statement 5. See Section 9.1 for a more detailed discussion about this phenomenon.*

Note that in the above example all diagnoses are so called single-diagnoses, i.e., diagnoses comprising only one component. The interpretation of a single-diagnosis is straightforward in such a way that the specified misbehavior of the system can solely be explained by the malfunction of a single component. Multiple-diagnoses on the other hand contain two or more components. In the case of multiple-diagnoses the incorrect system behavior can only be explained as the combination of all components in the diagnosis behaving incorrectly at the same time.

In principle, all potential diagnoses are of interest regardless of their cardinality. This is, because every diagnosis explains the malfunction of the system and it is assumed that by correcting all components within the diagnosis the expected system behavior can be achieved. Note that this is only correct for a perfect model. In case of FDMs diagnosis candidates can be computed, which in practice are not diagnoses (see Section 9.1). However, in practice smaller diagnoses (in particular single-diagnoses) are preferred over larger ones for the following reasons:

- Generally, smaller diagnoses seem to be more likely. Although (almost) all diagnoses explain the malfunction of the system, the majority of diagnoses are not correct diagnoses in the sense that they represent the system intended by the programmer. For instance, the assumption that all components of a given system behave abnormally is a diagnosis as far as the MBD approach is concerned. However, this diagnosis is not useful for transforming an incorrect system into a correct one. Since the probability of a single fault in a system seems higher than a fault affecting more components, smaller diagnoses are preferred over larger ones.

- A second, more practical reason is that fewer components are easier to replace or correct. Clearly, it is more economic to fix a single bug in a given system than to make multiple changes to the source code, if the outcome is the same. Furthermore, fixing only a few components is normally the more general solution, which is expected to hold for not just the current test-case. By fixing many components the risk is high that a given system is fitted to a single test-case too tightly so that it is not appropriate for the general case.

Finally, it should be noted that the process of computing diagnoses for a particular method as proposed in this chapter has always to be seen in the context of a single test-case. If an *ETFDM* is used as underlying model, the system description is only valid for this one test-case. Since different test-cases require

different *ETFDMs*, the test-case is defined by the evaluation trace used during the creation of the *ETFDM*. In contrast to *ETFDMs*, *DFDMs* are models, which take all possible run-time scenarios into consideration and thus implicitly cover all possible test-cases. Nevertheless, the process of computing diagnoses cannot be seen independently from a test-case. This is, because observations of variable occurrences can in general only be specified, if a whole evaluation trace or at least a concrete input/output pair is used during the debugging process (not during the creation of the model). Observations like *the value of variable occurrence vo is incorrect in all possible run-time scenarios* can hardly be stated in the general case. However, the concurrent use of multiple test-cases is possible, if the computation of diagnoses is slightly changed. Section 18.2 deals with the application of multiple test-cases.

Chapter 17

Building a Debugger

In this section we show how to build a debugging tool, which enables the user to use logical system descriptions of `Java` programs as introduced in Chapter 16 for efficient software debugging.

17.1 The debugging process

The combination of software models and MBD techniques on the one side and classic software engineering (SE) tools, such as GUIs and intuitive debuggers, on the other side has been proposed by [22] and more recently in [25]. The main idea behind this approach is to combine AI techniques as described in this work with existing SE tools and knowledge about efficient human-machine interactions in order to guide a user through a fault localization process in an optimal way. In this context possible requirements of a debugging tool are:

User-friendliness: A debugging tool should be as user-friendly and intuitive for the user as possible. This includes short learning times, logical steps and sequences of the individual procedures, a clear and understandable GUI, etc...

Response time: The response time of the system has to lie in a range which is acceptable for industrial applications, i.e., in the range of a few seconds for interactive applications.

Efficiency: The fault localization process itself has to be designed as efficiently as possible. This includes a minimum number of queries put to the user, an optimal bug candidate elimination, an efficient measurement selection algorithm, etc...

The debugger presented in this chapter is based on an iterative approach of computing diagnoses, reporting current bug candidates to the user, querying information from the user, and eliminating bug candidates. The debugging process can be described by the following phases:

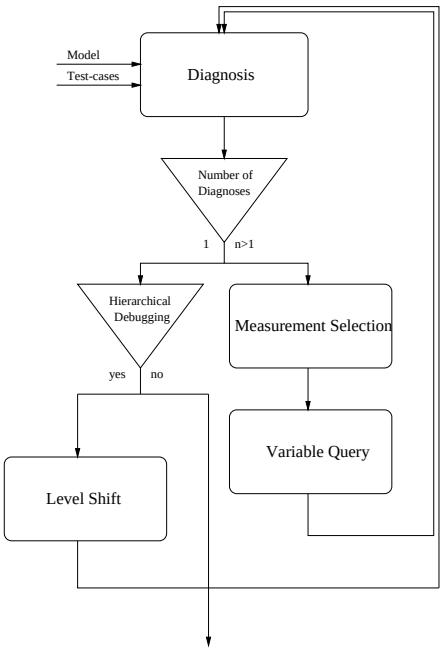


Figure 17.1: The debugging process

Initialization: Before we can start the debugging process we need a model of the program, which we are interested in debugging. This model has to be transformed into a logical description of the debugged source system. Furthermore, we need test-cases specifying the inputs of the debugged method together with the expected outputs. This information can directly be taken from the test phase of the software development process.

Diagnosis: An automatic debugger can then compute all diagnoses, which explain the malfunction of the system. These diagnoses highlight all potential software bugs provided that correct models and test-cases are available.

Measurement selection: As long as there exists more than one diagnosis, we select a measurement point within the `Java` system, whose validation reduces the number of potential diagnoses in an optimal fashion. The goal is to limit the search space of bug candidates by eliminating a maximum number of incorrect diagnoses in a minimum number of steps.

Variable Query: Once a measurement point is chosen, its value has to be measured. This can be done by querying the user for the correct value of the variable at the measurement point. The specification of additional observations reduces the number of diagnoses in the next step.

Hierarchical Debugging: If only one explanation for the faulty behavior of a system is present, the bug is located at the current hierarchy level. Debugging can then be terminated or led into the next level of the system

hierarchy.

Figure 17.1 shows the complete debugging process. Whereas the modeling of `Java` systems and the computation of diagnoses are discussed in previous chapters, the following sections deal with an efficient measurement selection algorithm, variable queries, and hierarchical debugging.

17.2 Measurement selection

The goal of an efficient measurement selection algorithm is to determine an optimal measurement point in the system description. After an additional observation is specified for a measurement point and added to the diagnosis system, the number of diagnoses should decrease as much as possible. In our case we want to determine a system connection, i.e., a variable occurrence of the `Java` system, whose specification reduces the number of diagnoses in an optimal way, regardless of the concrete value assigned to the measured variable.

One way to determine an optimal measurement point is to compute all possible observations, i.e., all possible values for all potential measurement points. For each connection/value pair all diagnoses then have to be recomputed and the measurement point, which on average (over all its potential values) eliminates the most diagnoses is chosen. Obviously, this is computationally very expensive and too slow for real-world applications.

Another way is to use algorithms, which estimate an optimal measurement point. Algorithm 17.2.1 is a simplification of the algorithm presented in [11] and uses entropy functions for all connections of the system description:

Algorithm 17.2.1

- For each diagnosis d of all current diagnoses D perform a simulation run and for each connection $l \in \mathcal{L}$ of the system description count the number of correct ($true_l$), incorrect ($false_l$), and unknown ($null_l$) predictions.
- For each connection compute its entropy as $E(l) = \sum_{x \in \{true_l, false_l, null_l\}} abs((|x|/|D|) * ln(|x|/|D|))$.
- Compute the next measurement point $mp \in \mathcal{L}$ by maximizing the entropy, i.e., $mp = l_i \in \mathcal{L} \mid E(l_i) = max(E(l_j)) \; \forall \, l_j \in \mathcal{L}$.

Example 17.2.1 *Consider the system description depicted in Figure 17.2. The system comprises 4 diagnosis components, i.e.,* $COMP = \{st1, st2, st3, st4\}$*, which are arranged in a linear chain. Therefore, the system contains 5 connections, i.e., input connection* $i1$*, output connection* $o1$*, and three internal connections* $l1$*,* $l2$*, and* $l3$*. We can now assume* $i1$ *to hold the correct input value and* $o1$ *to be faulty. Formally, we add* $ok(i1) \wedge nok(o1)$ *to the logical system representation. Clearly, this results in 4 possible diagnoses, namely all components of the*

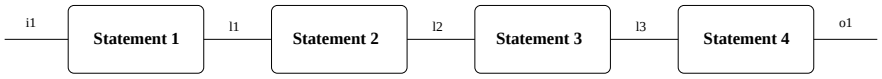


Figure 17.2: Measurement selection example 1

system. If we perform a simulation run for each diagnosis and count the number of values (`true`*,* `false`*, or* `null`*) produced for each of the connections* $l1$*,* $l2$*, and* $l3$*, we get the following result:*

Diagnosis	*Value*		
	l1	*l2*	*l3*
Statement 1	`null`	`null`	`null`
Statement 2	`true`	`null`	`null`
Statement 3	`true`	`true`	`null`
Statement 4	`true`	`true`	`true`

We can now compute the entropy for each connection using the Algorithm 17.2.1. This leads to the following result:

$$E(l1) = abs(0.25 * ln\,0.25) + abs(0.75 * ln\,0.75) = 0.56$$

$$E(l2) = abs(0.5 * ln\,0.5) + abs(0.5 * ln\,0.5) = 0.69$$

$$E(l3) = abs(0.75 * ln\,0.75) + abs(0.25 * ln\,0.25) = 0.56$$

From $mp = l_i \in \{l1, l2, l3\} \mid E(l_i) = max(E(l_j))\ \forall\, l_j \in \mathcal{L} = l2$ *it follows that our next measurement point has to be connection* $l2$*.*

To improve Algorithm 17.2.1 we make use of a heuristics. Consider a connection l_1, which is only linked to components that are not part of any diagnosis. If there exists another connection l_2 with the same entropy value as l_1, but with a connected component in at least one diagnosis, then l_2 is preferred over l_1 as a measurement point. The reason is that connections linked to components in at least one diagnosis are more likely to eliminate incorrect diagnosis candidates than other connections. The following example is designed to demonstrate this property of diagnosis systems.

Example 17.2.2 *Consider the system description depicted in Figure 17.3. Let us assume that input connection* $i1$ *is known to be correct and output connection* $o3$ *is incorrect. Then there are two diagnosis candidates, i.e., statement 1 and statement 4. Performing a simulation run on both diagnoses we get the following result:*

Diagnosis	*Value*		
	l1	*o1*	*o2*
Statement 1	`null`	`null`	`null`
Statement 4	`true`	`true`	`true`

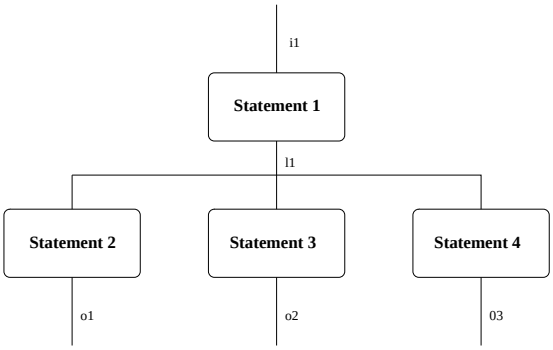


Figure 17.3: Measurement selection example 2

We find that the values of $o1$ *and* $o2$ *are equal to the value of connection* $l1$ *in all simulation runs. Therefore, the resulting entropies of* $l1$, $o1$, *and* $o2$ *have to be equal as well, which can easily be proven by the following calculation:*

$$E(l1) = abs(0.5 * ln\,0.5) + abs(0.5 * ln\,0.5) = 0.69$$
$$E(o1) = abs(0.5 * ln\,0.5) + abs(0.5 * ln\,0.5) = 0.69$$
$$E(o2) = abs(0.5 * ln\,0.5) + abs(0.5 * ln\,0.5) = 0.69$$

Using Algorithm 17.2.1 it seems that all measurement points have to be regarded as equally good as far as their ability to distinguish correct diagnoses from incorrect ones is concerned. This, however, is not true. If we take $o1$ *or* $o2$ *as the next measurement point, we have to distinguish two cases: (1) If* $o1$ *or* $o2$ *is specified to be incorrect, statement 4 is no longer a diagnosis and the bug is thus located in statement 1. (2) If* $o1$ *or* $o2$ *is specified to be correct, no diagnosis candidate can be eliminated. In the latter case the query of the value of connection* $o1$ *or* $o2$ *seems to be useless. If, on the other hand, we decide to use* $l1$ *as our next measurement point, one diagnosis is eliminated regardless of the specified value of* $l1$. *Thus* $l1$ *seems to be a superior measurement point than either* $o1$ *or* $o2$. *Using the heuristics stated above, we guarantee that in this example* $l1$ *gets selected as an optimal measurement point.*

17.3 Variable query

Once a measurement point *mp* is selected, the expected value at connection *mp*, i.e., the variable associated with connection *mp*, has to be read from some kind of oracle. Generally, there are two different approaches of specifying the expected value of *mp*:

- The value is known a priori. This, for instance, can be the case, if a detailed test specification exists, which not only contains a system's I/O behavior but also all or some of its internal states. In this case the expected values

of all variables or connections can be stored in a database and read into the debugging system on demand. Note that this approach is not optimal, because all values, which are known before the debugging session, could be specified during the initialization phase. This would speed up the debugging process as measurement selection steps and recomputations of diagnoses were no longer needed.

- The value can be queried from the user. In this case it is assumed that the user has enough knowledge about both, the programming language and the internal structure of the buggy program, to provide information about individual measurement points.

When using FDMs we are only interested in whether the value of a certain variable is correct or not. In contrast to value-based models we cannot handle concrete values or distinguish different degrees of incorrectness. Nevertheless, we can make use of an evaluation trace, if it exists. In such a case the value computed by the system is displayed to make it easier for the user to judge, whether this value is correct or not. However, the value itself is only needed for the sake of an intuitive GUI and not during the modeling and diagnosis process.

17.4 Hierarchical debugging

Once there is only one diagnosis left, the bug is located at the current level of the system hierarchy, i.e., at the current block level within the `Java` source code. If the bug is found in a statement containing sub-blocks, i.e., selection statements, loops, etc..., the debugger can automatically be led to sub-block level and locate the fault at the next level of the hierarchy. The diagnosis process at sub-block level works exactly like the process at a method's top-level and can thus be explained by Figure 17.1. However, before debugging at sub-block level can be started, some initialization has to be done:

- The debugger has to load the model of the sub-block, which should be debugged, and create a logical system description. Before the first diagnosis process, the theorem prover has to be loaded with the new system description. Note that all models constructed in Part II of this work are hierarchical models, which store all sub-models in the respective diagnosis components. In this case it is very easy to retrieve the model of sub-blocks. Another option is to create sub-models on demand, which leads to a shorter modeling but longer debugging time.

- The output connections of the new system description have to be initialized. This means that the user has to specify, whether the variable occurrences associated with the output connections are correct or not. In certain cases this can be done automatically, if the values of the outputs of the super-component can directly be used to initialize the sub-component. This, for instance is the case with selection statements and in the last iteration of loop statements. If an iteration of a loop other than the last one is to be

debugged, the super-component's values cannot be used and the user has to specify appropriate values.

Another question arising in the course of a hierarchical debugging process is, which branch or iteration of a loop or selection component to debug. Again, we can distinguish two different approaches:

- If no run-time information is used, the user has to guide the search process into a sub-block or branch manually. This is not an optimal strategy, because the user is not guaranteed to locate the bug once a wrong decision is made at a higher level of the hierarchy.
- The other option is to make use of evaluation traces. In such an approach the branch of a selection statement evaluated at run-time can be determined by looking at the run-time value of the statement's condition. The fault localization process can automatically be led into the correct branch. This is not possible for loop statements, where the iteration in which a bug appears first cannot be determined automatically in the general case. One approach is to let the user step through all iterations manually, but provide him with as much information as possible about the variable environments before and after individual iterations. The user can then enter the loop body once an unexpected behavior becomes observable.

17.5 Debugging method calls

In previous section we show how the fault localization process can be guided into a selection statement's branch or a loop body. However, there is yet another possibility of carrying on the debugging process after a single-diagnosis is computed. If a bug is located in a called method m, the debugging process can be guided into this method without restarting the whole debugging process. The debugging of method m works very much like the debugging of the calling method.

The inputs of method m are all assumed to be correct, whereas its outputs can be set to the values set for the component containing the method call in the calling method's system description. If an evaluation trace is needed, it can be taken from the evaluation trace of the calling method, which as a sub-trace contains the trace of the called method m. We can then start the diagnosis process exactly like for the calling method.

Chapter 18

Enhancements

This section describes some enhancements to the debugging process. In particular, the following topics are discussed:

- The design and implementation of an assertion language.
- The computation of diagnoses using multiple test-cases.

In the course of the JADE project both topics have not only been tackled theoretically, but also been integrated into the JADE debugging environment (see Chapter 19).

18.1 Assertions

The JADE assertion language helps to specify assertions and expected values of variables and expressions of Java programs. In particular, the language provides means to embed assertions either directly in the Java source code or to store them in separate files. The debugging environment is modified such that the assertions replace or supplement the traditional observation specification mechanisms of the debugger as described in Chapter 17. Consequently, the use of assertions makes it possible to run test-cases unattendedly. Also, multiple test-cases can be applied concurrently to improve diagnosis performance (see Section 18.2).

18.1.1 Assertion syntax

The syntax of the assertion language is chosen in a way that it can easily be integrated in existing Java programs. An important feature of the language is that the resulting program is still a valid Java program and can be parsed and compiled by existing Java tools. This is achieved by embedding the assertions within Java comments, similar to Java documentation comments (`/** ... */`). To distinguish assertions from ordinary Java comments and to allow for simpler parsing, the assertions must be delimited by the strings `/*@` and `@*/`.

Besides the delimiting tokens, an embedded assertion may consist of several parts. First, an assertion may be prefixed with the name of a test-case, to which

the assertion belongs. This allows the specification of assertions of several test-cases in a single file. If this part is not present, it is assumed that the assertion is valid for all test-cases. The main part of an assertion consists of a boolean expression, which should evaluate to `true`. Here, any valid `Java` expression can be used, provided that the semantics of the program is not altered.

The above mechanisms to specify assertions is sufficiently general to specify almost any reasonable assertion used in our software debugging environment. However, as the number of assertions and test-cases increases, the program soon becomes cluttered with assertions. To avoid this, the assertions can be put in an extra file and be associated with parts of the program. This is done by introducing labels which denote parts of the program, where assertions will be inserted later on. Labels are specified similarly to assertions, but without the expression part. In the separate file, the assertions are prefixed by the label to which the assertion belongs.

The assertions and assertion labels can be inserted into the program at several places: Pre- and post-conditions of methods are specified by inserting assertion comments immediately before or after the method's body. Further assertions can be inserted within methods after each statement. Note that assertions at expression level cannot be specified in the current implementation. However, the specification of assertions at statement level is sufficient for our current purposes.

All syntactical enhancements, which are introduced in the context of the assertion language, are defined by the following grammar rules (EBNF notation):

AssertionComments	::=	{ AssertionComment }
AssertionFile	::=	{ AssertionLabel : [Assertion] }
AssertionComment	::=	/*@ (AssertionLabel | Assertion) @*/
AssertionLabel	::=	Name
Assertion	::=	[Name :] (Expression)

The non-terminals *Name* and *Expression* are defined as in the `Java` Language Specification [16] and denote simple identifiers and `Java` expressions. The integration of the above language definition is done through two modifications of the `Java` parser, resulting in the following production rules (changes are emphasized):

MethodBody	::=	*AssertionComments* Block *AssertionComments*
BlockStatement	::=	(LocalVariableDeclarationStatement
		| ClassOrInterfaceDeclaration
		| [Identifier :] Statement) *AssertionComments*

The separate assertion file is parsed as specified by the grammar rule given above (non-terminal *AssertionFile*) using a modified version of the `Java` parser.

Example 18.1.1 *Examples of pre- and post-conditions following the grammar rules as described above can be found in the source code listings in Figures 18.1 and 18.2.*

18.1.2 Semantic restrictions

Although the grammar given in the previous section allows any valid `Java` expression to be used as assertion expression, some semantic restrictions have to be followed. Note that these restrictions are not checked by the system. It is assumed that all assertions are well-formed according to these rules.

- First, any expression used as assertion must be executable at the point of specification. Pre- and post-conditions must be executable before the first and after the last statement of the method, respectively. Note that post-conditions must be executable immediately after the evaluation of the return expression of each return statement of the method.
- The expression must not alter the behavior of the program. In particular, no values may be assigned to variables and no methods with side-effects may be called.
- The expression must evaluate to a boolean value and it must not throw an exception.
- The names of test-cases and assertion labels must be globally unique. They are not local to methods, classes or packages.

The assertion language as defined above can be used in combination with various software models, e.g., FDMs, value-based models, etc... However, in this work we focus on its use together with FDMs for concrete debugging problems. In order for the assertion language to be applicable to FDMs some further restrictions have to be met, which can be described as follows:

- The syntax of expressions is restricted such that only one variable of the method is used. Otherwise, the assertion's associated variable cannot be determined automatically.
- Variables and fields may be used in expressions as far as they are used in the enclosing method. Unused variables and fields may not be included in assertions.
- The form of assertion expressions used in pre-conditions of methods is limited to $v == c$, where $v \in VARS$ denotes a parameter or global variable and $c \in CONSTANTS$ represents a constant expression.
- All input parameters and used global variables must be assigned a value for each test-case. Otherwise, the program cannot execute and the assertions cannot be compared with the computed values.

Assertions that do not respect these restrictions are assumed to be correct and are ignored.

18.1.3 Using assertions for debugging

This section describes the current implementation of the assertion mechanism and its integration into the debugger in the context of FDMs. When the parser recognizes an assertion comment, the assertion is added to the current node of the parse tree (representing a statement or a method declaration). When separate assertion files are parsed later on, the assertions therein are added to the node of the parse tree that contains the same assertion label as the assertion. Note that the current implementation requires the assertion labels to be globally unique.

A further requirement is that every test-case must be present in the method's pre-condition in order to be applicable to the method. Otherwise the system does not include the test-case in the list of defined test-cases. If the method has formal parameters, this is not considered a restriction, as every test-case must fully specify the values of the formal parameters. If no parameter variables are present, an empty pre-condition (containing the constant `true` as condition) can be inserted.

When assertions are used for debugging in combination with FDMs, the modeling process itself is not modified, i.e., assertions are not incorporated into the FDMs, but only used during the debugging process. Moreover, the model building process of the program is separated from the modeling of the assertion expressions. The algorithm of using assertions for debugging can be summarized as follows:

Algorithm 18.1.1

1. Build a model of the program as before, but ignore all assertions.

2. Generate an instrumented version of the program, including all assertion expressions. The assertions are converted to equivalent `Java` code and their result is written to a log file for later use. Note that this step can be fairly complex for return statements and constructor invocations, as the expressions have to be duplicated and wrapped in function calls. Pre-conditions of methods are handled specially, as they are converted into statements assigning the specified value to the corresponding parameter variable.

3. Read the log and extract all values computed by the program and all values resulting from assertion components. For all assertions that are associated with exactly one variable, compare the computed value with the value of the assertion expression and set an observation accordingly. The connection representing the variable occurrence, say vo, is observed as $nok(vo)$ iff the values are different. For assertions that evaluate to `true` no observations are specified, since this is impossible for general expressions. For example, consider the assertion $x > 5$. If the assertion evaluates to `true` it is still not guaranteed that the value of x is correct.

4. Perform diagnosis as before.

Assertions are introduced to improve the debugging process without directly influencing the used FDMs. The advantages of the use of assertions can be summarized as follows:

- General observations can be specified by the user either directly in the source code or in a separate assertion file. These observations can be used during the debugging process to eliminate incorrect diagnosis candidates and to minimize the amount of user interaction.
- Individual test-cases can be defined, which can then be debugged either interactively as described in Chapter 17 or off-line without the attendance of the user. The latter application is a prerequisite for the use of multiple test-cases in a single debugging session, which is discussed in the next section.

18.2 Using multiple test-cases

In the previous section we describe the use of an assertion language in order to enhance the debugging potential of an automatic debugging tool. Another enhancement building on the notion of assertions is the extension of the model-based theory presented in Chapter 3 to the concurrent application of multiple sets of observations, i.e., the use of multiple test-cases, in a single diagnosis step. Theory and implementation of this approach are discussed in this section.

18.2.1 Extending MBD

So far we are dealing with diagnosis problems of the form $(SD, COMP, OBS)$ (see Chapter 3), which include a single set of observations, i.e., OBS. In the context of software debugging we are speaking about a single test-case, which comprises the observed values (`true` or `false`) of various variable occurrences. It is important to note that so far it is not possible that multiple observations of the same variable occurrence are incorporated into a diagnosis problem. [43] propose an approach of extending the standard MBD definitions in a way that multiple test-cases can be handled simultaneously. Let OS be a set of test-cases. More formally, we define:

Definition 18.2.1 *A collection of test-cases of a given method, i.e., OS, is a finite set of observations, each of which is itself a finite set of first-order sentences. The triple $(SD, COMP, OS)$ is called a diagnosis problem for the system $(SD, COMP)$ with multiple test-cases OS.*

In analogy to Chapter 3 we are now looking for all sets of components, whose malfunction explains the incorrect behavior of the system. In other words we want to compute all diagnoses for a given diagnosis problem with multiple test-cases. Formally, we write:

Definition 18.2.2 ([43]) *A diagnosis Δ for a diagnosis system $(SD, COMP)$ using multiple test-cases OS is a subset of $COMP$ such that $\forall_{OBS \in OS}\,(SD \cup OBS \cup \{AB(C) | C \in \Delta\} \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \not\models \bot)$.*

Similar to the standard MBD definitions we define a conflict as the dual concept of a diagnosis. A conflict specifies a set of components, which (given the model and a set of test-cases) cannot all work as expected at the same time. This means that at least one component exhibits unexpected behavior in at least one of the specified test-cases. More formally, we write:

Definition 18.2.3 ([43]) *A conflict set for* $(SD, COMP, OS)$ *is a set* $CO \subseteq COMP$ *such that* $\exists_{OBS \in OS} (SD \cup OBS \cup \{\neg AB(C) | C \in CO\} \models \bot)$.

We now compute all diagnoses of a given diagnosis system with multiple test-cases by applying the hitting set algorithm (see [38, 17]) to all conflict sets produced by all test-cases in OS. In particular, the relationship between diagnoses and conflicts is stated by the following modification of Theorem 3.1.1:

Theorem 18.2.1 *The set* $\Delta \subseteq COMP$ *is a (minimal) diagnosis for* $(SD, COMP, OS)$ *iff* Δ *is a (minimal) hitting set for the collection of conflict sets produced by all test-cases* $OBS \in OS$.

We see that the basic technique is still the same with the only difference that now multiple test-cases are used to compute conflicts. Clearly, the more test-cases we use, the more conflict sets we can possibly get. In turn, a higher number of conflicts as input to the hitting set algorithm means a lower number of resulting single-diagnoses and thus a more accurate diagnosis process.

18.2.2 Computing diagnoses using multiple test-cases

The techniques described in the previous section can easily be incorporated into an existing debugging tool based on MBD algorithms. This section shows how diagnoses can be computed, if multiple test-cases are used in combination with *DFDMs* and *ETFDMs*. We also discuss, in which cases an improvement in the resulting number of diagnoses can be expected by adding new test-cases to the system.

***DFDM*:** If multiple test-cases are used with a *DFDM* as the underlying model, the goal is to compute all diagnoses arising from different observations of the same system description. In other words, we specify multiple test-cases for a given method m, but the underlying model $DFDM_m$ does not change. This is, because a *DFDM* covers all possible run-time scenarios and can thus not be adapted to individual test-cases. In particular, Algorithm 18.2.1 can be used to compute all diagnoses for the diagnosis problem $(SD, COMP, OS)$, if $SD = DFDM_m$ holds:

Algorithm 18.2.1

1. Compute the underlying model as $SD = DFDM_m$.
2. Specify a set of test-cases, i.e., OS, for a given diagnosis system $(SD, COMP)$.

```
void multipleTestcases1(int i, int j)

   /*@ t1:(i==0) @*/ /*@ t1:(j==1)@ */
   /*@ t2:(i==1) @*/ /*@ t2:(j==0)@ */
{
1.   int k = 5;
2.   int x = i*k;
3.   int y = j*k;

   /*@ t1:(x==0) @*/ /*@ t1:(y==3)@ */
   /*@ t2:(x==3) @*/ /*@ t2:(y==0)@ */
}
```

Figure 18.1: Example method *multipleTestcases1(int i, int j)*

3. For each test-case $OBS \in OS$ use a theorem prover to compute all conflicts arising from the diagnosis problem $(SD, COMP, OBS)$, i.e., COS_{OBS}.
4. Compute the set of all conflicts of all test-cases as $COS = \bigcup_{OBS \in OS} COS_{OBS}$.
5. Apply the standard hitting set algorithm to COS to compute all diagnoses of $(SD, COMP, OS)$.

From Algorithm 18.2.1 it follows that the same *DFDM* is used for the computation of conflict sets, even, if multiple test-cases are used. Therefore, an improvement of the diagnosis process can only be expected, if different inputs of the same *DFDM* produce different conflict sets. This behavior is demonstrated by the following example:

Example 18.2.1 *Consider method multipleTestcases1(int i, int j), which is depicted in Figure 18.1. Assume that statement 1 contains a bug and should read* `int k = 3;`*. Then depending on the two specified test-cases we get the following scenarios: (1) When running test-case* $t1$*, the value of* x *is correct, whereas the value of* y *is incorrect (5 instead of 3). (2) With test-case* $t2$ *the opposite is true. Variable* x *is incorrect (should be 3 instead of 5) and* y *is correct. Therefore, we get two different conflict sets for the two program runs, i.e.,* $\{s1, s3\}$ *for test-case* $t1$ *and* $\{s1, s2\}$ *for test-case* $t2$*. By applying the hitting set algorithm we get a single fault location in statement line 1. Obviously, in this case using two test-cases outperforms the debugging with just one observation.*

***ETFDM*:** The situation is somewhat different, when *ETFDMs* are used as underlying models. This is, because *ETFDMs* only model a single program run of method m and thus directly correspond to a particular test-case. If we now use multiple test-cases, we also have to use multiple *ETFDMs* at the same time in order to compute a maximum amount of conflict sets. Note that we must

not compute a combined model of all *ETFDMs*, because by doing so we would lose information about the individual program runs. Therefore, we rather use all *ETFDMs* separately to compute conflict sets, which can then be brought together during the hitting set algorithm. Algorithm 18.2.2 shows how diagnoses are computed using multiple test-cases in combination with *ETFDMs*:

Algorithm 18.2.2

1. Specify a set of test-cases, i.e., OS, for a method m
2. For each test-case $OBS \in OS$ compute $SD_{OBS} = ETFDM_m^{OBS}$. Use a theorem prover to compute all conflicts arising from the diagnosis problem $(SD_{OBS}, COMP, OBS)$, i.e., COS_{OBS}.
3. Compute the set of all conflicts of all test-cases as $COS = \bigcup_{OBS \in OS} COS_{OBS}$.
4. Apply the standard hitting set algorithm to COS to compute all diagnoses of $(SD, COMP, OS)$.

Using Algorithm 18.2.2 different conflicts are produced for different evaluation traces and their corresponding *ETFDMs*, which in practice can lead to a smaller amount of resulting diagnoses. This principle is clarified by the following example:

Example 18.2.2 *Consider method multipleTestcases2(int x, int y), which is depicted in Figure 18.2. This method computes four different functions $z = f(x, y, c)$ depending on the values of x, y. Note that a single bug is installed in statement line 2, which should read* `c=3;`*. The four test-cases specified by the pre- and post-conditions in Figure 18.2 correspond to four different control flow paths through multipleTestcases2(int x, int y), in each of which exactly one of the four* ***if*** *statements in lines 3 to 6 is executed. Each of the four resulting ETFDMs contains FDs for statements 2 and 7, but only one of the four* ***if*** *statements. If we assume that variable z is observed to be incorrect after the execution of the whole method, four conflicts are computed: $\{s2, s3, s7\}$, $\{s2, s4, s7\}$, $\{s2, s5, s7\}$, $\{s2, s6, s7\}$. Applying the hitting set algorithm, we only get two possible minimal diagnoses, i.e., $\{s2\}$ and $\{s7\}$, instead of 7 in case of the DFDM. This is, because by using multiple test-cases all four* ***if*** *statements can be eliminated as possible culprits.*

Finally, we shortly describe how the overall debugging procedure changes, if the handling of multiple test-cases is incorporated into a debugging system. First, the debugging interface has to provide means of allowing the user to specify multiple test-cases for a single method. This can either be done interactively or more elegantly by the use of an assertion language. The `JADE` debugger currently accepts the input of multiple test-cases through the specification of pre- and post-conditions or assertions as defined in Section 18.1. Second, the whole debugging process has to be adapted to multiple test-cases. The following two strategies can be used: (1) Diagnoses are computed off-line, i.e., without any user interaction.

```
      int multipleTestcases2(int x, int y)

        /*@ t1:(x==2) @*/ /*@ t1:(y==2) @*/
        /*@ t2:(x==2) @*/ /*@ t2:(y==-2) @*/
        /*@ t3:(x==-2) @*/ /*@ t3:(y==2) @*/
        /*@ t4:(x==-2) @*/ /*@ t4:(y==-2) @*/
      {
        int i1,i2,z,c;
1.      z = 0;
2.      c = 5; // should be c = 3;
3.      if (x >= 0 & y >= 0) {
3.1.      z=2*x*x + y*y + 3*x*y + c; }
4.      if (x >= 0 & y < 0) {
4.1.      z=x*x + 7*y*y + -2*x*y + c; }
5.      if (x < 0 & y >= 0) {
5.1.      z=x*x + -3*y*y + 6*x*y + c; }
6.      if (x < 0 & y < 0) {
6.1.      z=-4*x*x + y*y + -1*x*y + c; }

        /*@ t1:(z==27) @*/
        /*@ t2:(z==-21) @*/
        /*@ t3:(z==-29) @*/
        /*@ t4:(z==-13) @*/
7.      return z;
      }
```

Figure 18.2: Example method *multipleTestcases2(int x, int y)*

This works well in combination with the specification of multiple test-cases by using an assertion language as described above. Nevertheless, exact fault locations cannot always be found due to a possible lack of observations. (2) If an interactive debugging strategy is to be employed, it has to be modified to accept inputs for multiple test-cases in each step. This involves a changed measurement selection algorithm and new forms of variable queries. Such an approach seems quite complex and is possibly no longer intuitive for the user. This is, why currently the `JADE` debugger only features the use of multiple test-cases in an off-line mode.

Chapter 19

The JADE Debugging Environment

The JADE debugging tool is a prototype debugging environment, which has been developed as one major activity of the JADE project (see Chapter 4). It is a GUI tool, which enables its user to model and debug Java systems.

19.1 The JADE system

The whole JADE system is implemented in Smalltalk in the course of the JADE project. Figure 19.1 shows all modules of the JADE system and their interdependencies. These modules can shortly be described as follows:

Parser: The JADE parser transforms valid Java programs to an internal parse tree representation and has to be seen as the base component for all modules working with the source code structure of an analyzed Java system. The Java parser accepts the full syntax and semantics of a given Java program and can thus handle not only small user-defined programs, but also larger applications including the full range of Java system classes. Currently, the Java parser understands all source code elements defined in version 2 of the Java Language Specification [16]. Note that the parser has been modified in a way that it is able to handle in-line assertions and assertions defined in special assertion files (see Section 18.1). In both cases all assertions are stored directly in the internal parse tree and can be accessed by other modules.

MBD: The model-based diagnosis module provides all data structures necessary for the creation of system descriptions as defined in Chapter 15. Furthermore, it contains all algorithms, which are needed for the computation of conflicts and diagnoses including a standard theorem prover. It also provides a measurement selection algorithm as proposed in Section 17.2.

FDM: The functional dependency modeling module creates *ETFDMs* and *DFDMs* of given Java programs as described in Part II of this work. It

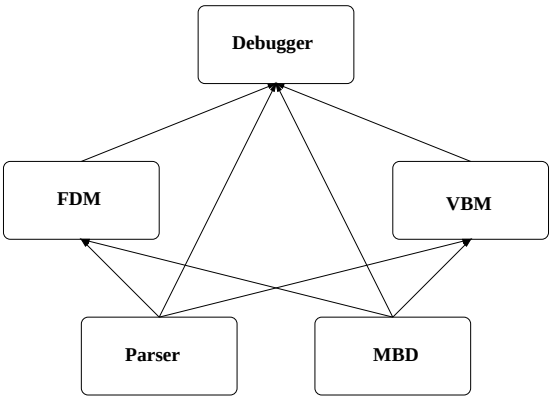


Figure 19.1: JADE modules

builds on the internal parse tree representation, which is returned by the parser module. It provides a set of hierarchical diagnosis components, which are directly used for debugging. Currently, the FDM module handles a large subset of the Java programming language. Source code structures, which are currently not or only partly handled by the JADE system are:

- Labeled, **break** and **continue** statements
- **Switch** statements
- Exception handling (keywords **throw** and **try**)
- Synchronization of multiple threads (keyword **synchronize**)
- Inner classes
- Special cases, such as reflexion and method arrays

VBM: Similarly to the FDM module, the value-based modeling module creates value-based models building on the internal parse tree representation returned by the parser module. However, the creation of these models and its application to debugging is out of the scope of this work (see [32]).

Debugger: The debugging module is based on all modules described above and brings together the software models and concrete diagnosis algorithms as provided by the MBD module. The exact features of the debugger module are described in the following sections.

19.2 The JADE debugger

The main constituent of the JADE debugger module is the JADE debugging environment. Figure 19.2 shows the main GUI of the JADE debugger after the source code of class *Point* (see Figure 5.1) is loaded into the system. The JADE debugging environment is a debugging tool for the efficient localization of source code

faults in Java programs. It is designed to guide the user through the debugging process in an optimal fashion as far as information queried from the user and visualization techniques are concerned. The main features of the JADE debugging environment in combination with the use of FDMs are:

Parsing: As already mentioned the JADE debugger makes use of the parser module. The debugging tool can parse user-defined programs and system classes and transform them into an internal parse tree representation. Assertion files (see Section 18.1) can be added to the parse tree.

Modeling: The next step in every debugging procedure is to compute a model of the Java program. *ETFDMs* and *DFDMs* can be created from the debugging tool via a call to the FDM module. The JADE debugger contains all algorithms for the computation of the *SFDM* of the underlying model, if requested by the user.

Debugging: The main functionality of the JADE debugging tool is to compute diagnoses of the analyzed Java program. This is done by following the principles of MBD as discussed in Chapters 3 and 16. In particular, there are three different diagnosis modes:

1. The computation of all diagnosis candidates of a Java program in the context of a single test-case (see Chapter 16).
2. The computation of all diagnosis candidates of a Java program in the context of multiple test-cases (see Section 18.2).
3. The exact localization of a single bug in a Java program in the context of a single test-case. This is done by employing an interactive and iterative debugging process, which successively uses more information and thus eliminates incorrect diagnosis candidates until eventually a single bug location is identified. This debugging mode is described in more detail in Chapter 17 and the following section.

Note that an interactive debugging process in combination with multiple test-cases is not implemented in the JADE debugger. This is mainly, because it seems to be too complicated for the user to specify variable values belonging to different evaluation traces in one debugging run. An efficient and intuitive algorithm for an interactive debugging strategy involving multiple test-cases is therefore left to future research projects.

Code instrumentation: The JADE debugging tool makes use of a code instrumentation component, which computes evaluation traces. These traces are used during the whole debugging process in two ways: (1) A trace can be created by the debugger and then be used in order to create an *ETFDM* of the analyzed Java method. If no evaluation trace is created before the modeling, only *DFDMs* can be computed. (2) A trace can be used during a debugging session to display current variable values to the user during variable queries (see Section 17.3) or to automatically guide the user into sub-models in case of hierarchical debugging (see Section 17.4). Since the

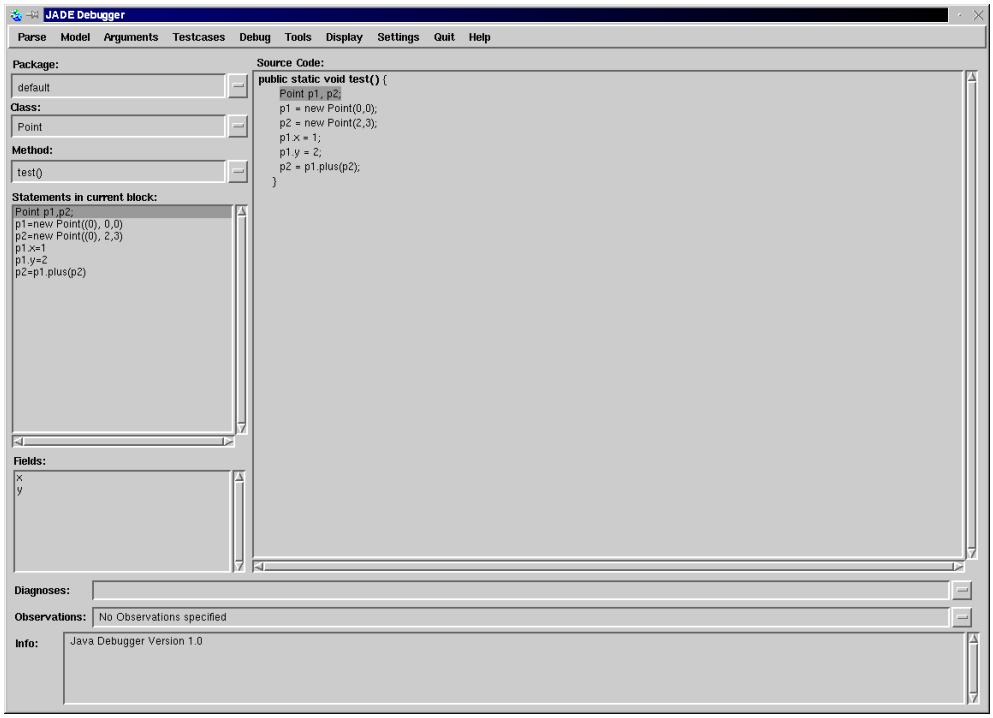


Figure 19.2: The JADE debugger main window

latter use of traces is not affecting the modeling process, it can be applied to both model types, *ETFDMs* and *DFDMs*.

Database controller: The JADE debugger provides the option of removing individual methods from the modeling and debugging scope. These methods are stored in an internal database and ignored during the modeling process. This is helpful, if the user wishes to use default models (see Section 12.2) of certain methods.

19.3 An interactive debugging session

The procedure of an interactive and iterative debugging process, which exactly locates a single bug in a given Java program, has already been discussed in Chapter 17 (see Figure 17.1). The following paragraphs show how the fault localization process is implemented in the JADE debugger in detail. Note that it is always assumed that a single bug is the source of the incorrect output produced by the program under consideration. This is, because at certain points during a debugging session we have to guide the debugger into a statement's sub-block or the body of a called method. In case of multiple faults this would pose an indeterministic problem, which has not yet been tackled by the JADE project. If

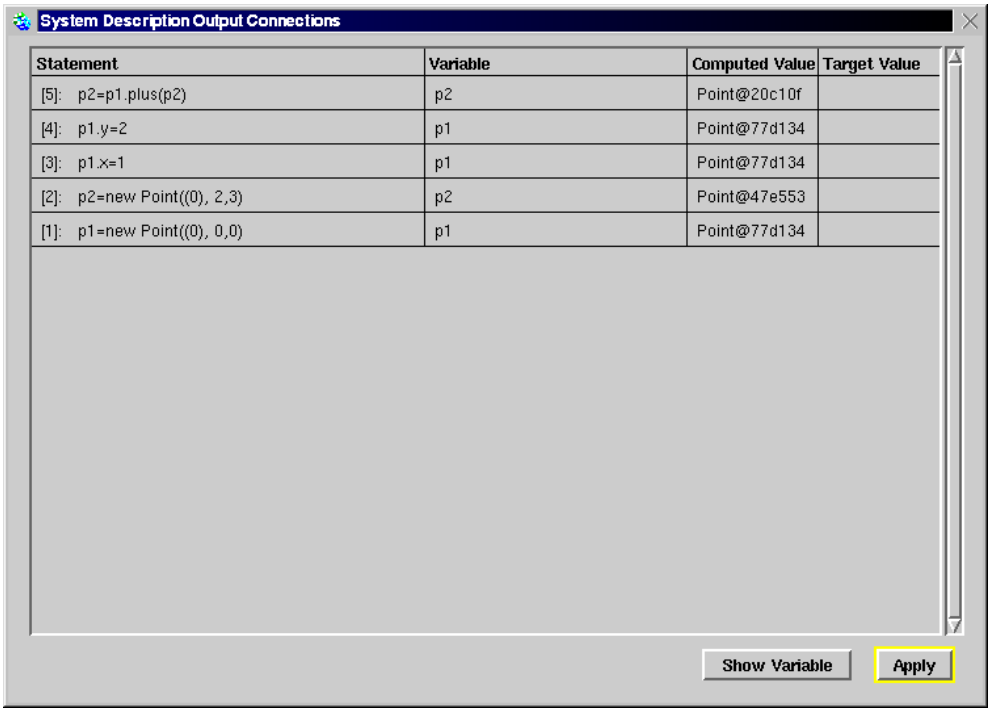


Figure 19.3: Specifying initial observations

we come back to the iterative debugging sequence as depicted in Figure 17.1, we encounter the following phases:

Specification of observations: As already discussed, diagnoses for a given system can only be computed, if observations contradicting the behavior described by the logical model are specified. We therefore need a successful test-case containing at least one variable in an incorrect state. The `JADE` debugger supports the following ways of specifying initial variable observations:

- Normally, at the start of each interactive debugging session the user is asked to provide initial values for certain variables. The GUI used by the `JADE` debugger for querying these observations is depicted in Figure 19.3. For each variable v the user can either state whether the value of v is correct or incorrect by entering `true` (`false`) or `ok` (`nok`) in the respective target value field or specify the exact target value. In the latter case the `JADE` debugger internally compares the target value with the value computed by the code instrumentation module to generate observations of the form $ok(v)$ or $nok(v)$. It is further possible to choose, whether all variable occurrences of the system or only the ones associated with output connections are displayed. Figure 19.3 shows the GUI used by the `JADE` debugger to specify initial observations for method *test()* of class *Point* (see Figure 5.1). Here, the *SFDM* of

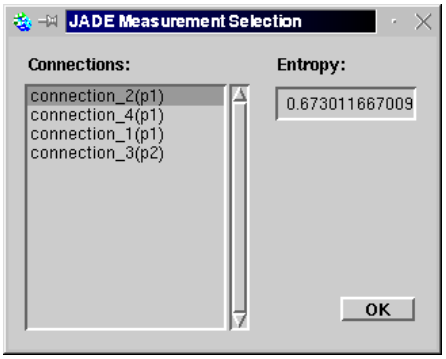
test() is used as an underlying model and all system connections are initially put to the user.

- An alternative way of specifying observations is the use of assertions and pre- and post-conditions (see Section 18.1). The user can either specify general assertions, i.e., assertions not defined for a particular test-case, or assertions, which are only valid for a single test-case t. Whereas the latter are only considered during the debugging of t, general assertions are used in all diagnosis sessions.

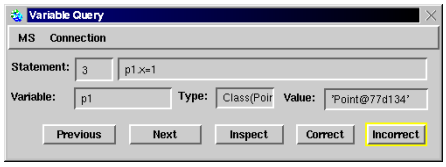
Computation of diagnoses: Once observations are specified, diagnoses are computed. When using the JADE debugger, the user can set the maximum size and maximum number of diagnoses to be produced. The maximum diagnosis size determines the maximum amount of statements in each diagnosis candidate. If only single-diagnoses are to be computed, this parameter should be set to one. The maximum number of diagnoses specifies how many diagnoses are computed. Both parameters are used to limit the maximum computation time of the diagnosis algorithms.

Measurement selection: The JADE system uses exactly the measurement selection algorithm proposed in Section 17.2. In case of multiple connections having the same entropy the heuristics discussed in Section 17.2 is used to distinguish connections linked to components within at least one diagnosis from other connections. If two connections linked to components within at least one diagnosis have the same entropy, one connection is randomly chosen. Figure 19.4 (a) shows the results of a measurement selection process during the debugging of method *test()*. All system connections, whose value has not yet been specified, are displayed with the top connection showing the highest entropy value. The exact entropy of the currently marked connection is presented in the *entropy* field on the GUI's right-hand side.

Variable query: Once an optimum measurement point within the system description is selected the user has to provide values for that measurement point. In our case a measurement point is a system connection, which can non-ambiguously be associated with a particular variable occurrence of the debugged method. Therefore, the user has to state whether the value of the variable occurrence is correct or not. Furthermore, the user has the option to reject a proposed measurement point and evaluate the next best variable occurrence. This can be very helpful in cases, where the user has a priori expectations about the possible location of a bug. Figure 19.4 (b) shows the GUI used by the JADE debugger for a single variable query. The buttons `Previous` and `Next` can be used to select the different measurement points in an ordered list starting with the variable occurrence with the highest entropy. Note that in such a way only one variable value can be specified in each step of the debugging procedure. Alternative GUIs similar to the one used to specify initial observations (see Figure 19.3) could be used to allow for the evaluation of multiple variable occurrences in one step.



(a) Measurement selection



(b) Variable query

Figure 19.4: Selection and evaluation of a measurement point

Further computations: The new observations read in through the variable query GUI are now expressed in logical sentences and added to the knowledge base. Then new diagnoses are computed by the JADE debugger. This process can be continued until eventually only one diagnosis is left, which explains the misbehavior of the analyzed method. In this case the bug is located at statement level. The debugging process can now either be terminated by the user or guided into a sub-block or sub-expression of the buggy statement. The latter approach is possible, because all used models are hierarchical models. The exact procedure of hierarchical debugging with the JADE debugger is described in Section 19.4.

19.4 Hierarchical debugging

The JADE debugger makes use of a hierarchical debugging strategy. Once a single bug is determined at statement level, the debugging process can be guided into a sub-block of the buggy statement (in case of loop and selection statements) or into a called method (in case of a method call). In the following we shortly describe how the JADE debugger handles hierarchical debugging in the case of **if** statements, **while** loops, and method calls. Note that other selection and loop statements can be handled very similarly to **if** and **while** statements, respectively. If a fault is located in a loop or selection statement, it is possible that the statement is incorrect, altogether. For example, an **if** statement could be changed to a **while** statement. However, in this case hierarchical debugging is not feasible. In the following paragraphs we assume that the type of the statement containing a bug is correct and the fault can therefore be found in one of its sub-structures.

If statements: If a fault is located within an **if** statement, we know that either the statement's condition or one of its branches have to be buggy. The JADE debugger therefore has to guide the debugging process into one of the statement's

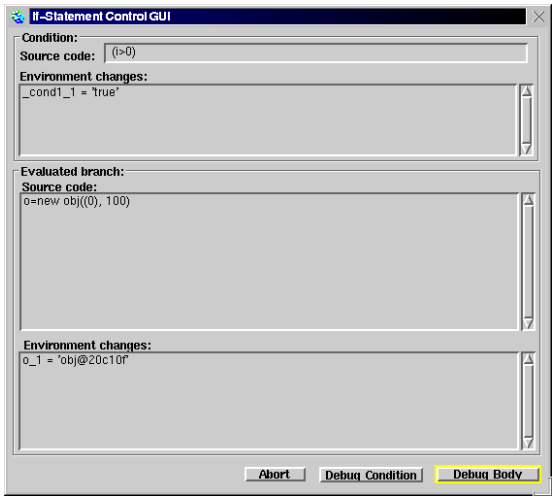


Figure 19.5: Debugging **if** statements

sub-structures depending on information provided by the user. In particular, the following two approaches are used in the JADE debugging tool:

- If no run-time information is used during the debugging session, the user has three options: (1) State that the bug is located in the condition, (2) debug the then-branch, or (3) debug the else-branch. Whereas the first case leads to a new debugging process at expression level, which is currently not implemented in the JADE system, the latter two cases start a new interactive debugging process as depicted in Figure 17.1 at sub-block level. Since no evaluation trace is used, the user gets no feedback about the condition's evaluation result. Thus, it seems doubtable whether the user can provide enough information to properly continue the debugging process.

- If an evaluation trace is available, the branch executed at run-time can be determined automatically. Therefore, in this case the user only has two options: (1) Mark the statement's condition as buggy or (2) lead the debugging process into the branch executed at run-time. In particular, the evaluation result of the condition is displayed to the user, who has to decide, whether the fault lies within the condition (in case of an incorrect evaluation value) or in one of the statements of the branch executed at run-time. This approach is superior to the first one, because the user has less options with a higher amount of information. Figure 19.5 shows the GUI used by the JADE debugger to debug incorrect **if** statements. The GUI shows the **if** statement in line 1 of method *if1(int i)* (see Figure 10.1), which has been detected as buggy with an input value of $i = 1$. Note that all run-time variable changes of statement 1 are displayed in order to support the user in his decision making.

While statements: If a fault is located in a **while** loop, the strategy is similar to the debugging of incorrect selection statements. We know that the fault can either be found in the loop condition or its body. In the latter case the user has to provide enough information to continue the debugging process at sub-block level. The following two approaches are implemented in the `JADE` debugging environment:

- If no run-time information is used during the debugging process, the user has to decide whether the fault can be found in the loop condition or to continue debugging in the loop body. Again, this seems to be very difficult, if the evaluation value of the condition is unknown. Note that in this case the user cannot specify a particular iteration of the loop, in which the fault gets observable for the first time. This is due to the following reasons: If a *DFDM* is used for debugging, the sub-models for all possible iterations are the same. The usage of *ETFDMs*, on the other hand, is not feasible without evaluation traces.

- If an evaluation trace is used, the user has to do the following: (1) Step through the execution of the loop iteration by iteration and find the first iteration, in which the bug becomes observable. (2) Specify, whether the bug appears in the loop condition or the loop body. In the latter case the debugging process is continued in the body. Note that the selection of the first incorrect iteration has to be done for the following reasons: If a *DFDM* is used, the body models for all iterations are the same, but the correct sub-trace of the used evaluation trace has to be determined for further debugging. If an *ETFDM* is used, the exact iteration has to be known to determine the correct sub-model, since there exist different models for different loop iterations. Furthermore, the correct sub-trace has to be computed as in the case of *DFDMs*. Figure 19.6 shows the GUI used by the `JADE` debugger to debug incorrect **while** statements. Here, method *while1(int i)* (see Figure 10.3) is debugged with the evaluation trace computed for an input value of $i = 1$. Once the **while** loop in statement 5 is found to be buggy, the user can step through the variable environment changes of the loop condition and its body in all iterations performed at run-time. This can be done by pressing one of the buttons in the upper list (`First`, `Previous`, `Next`, or `Last`). The buttons in the lower button list can be used to direct the debugging process to the loop condition or the loop body.

Note that incorrect **do** and **for** loops can be debugged in the same way. The `JADE` debugger uses exactly the same GUIs for these loops after the loop transformation algorithms described in Section 10.4.5 have been applied.

Method calls: When a method call is found to be the culprit of the misbehavior of a given system, two cases have to be distinguished:

1. The bug lies in the calling method. This is the case, if (1) the scope or part of the scope of the method call are incorrect, (2) one or more of the

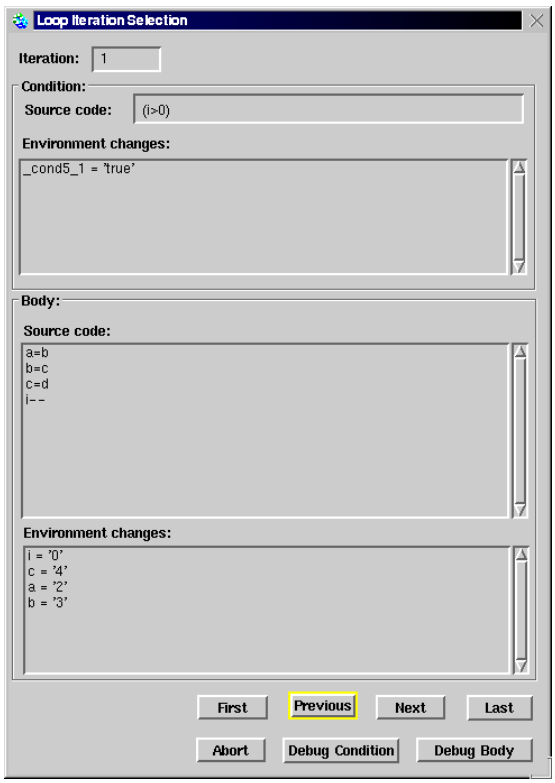


Figure 19.6: Debugging **while** statements

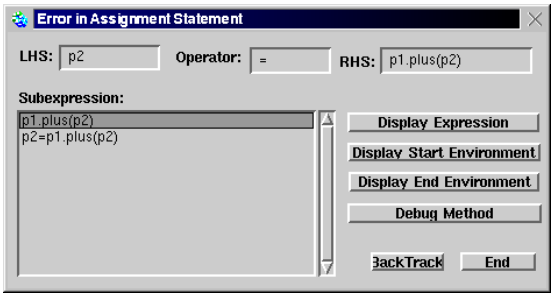


Figure 19.7: Stepping into an incorrect method

method call's arguments are incorrect, or (3) the wrong method is called. The latter case includes all scenarios, where an incorrect method signature can be found in the source code, e.g., an incorrect method name, a wrong number of arguments, etc... Since in all three cases the bug is located in the calling method, no further debugging in the called method has to be performed. Moreover, the method call in the calling method has to be changed in order to correct the fault.

2. The bug lies in the called method. This is the case, if the called method returns an incorrect value or exhibits incorrect behavior through side-effects. In this case the called method has to be debugged by leading the fault localization process into the faulty method. In the called method the debugging process works exactly like in the calling method. The initial specification of output observations in the called method's system description can be obtained automatically, because all known values of output connections of the buggy method call components can directly be used inside the called method.

The `JADE` debugger currently only features statement level debugging, which introduces a new problem, when method calls to buggy methods appear at expression level. At the moment the `JADE` tool displays the buggy statement in the calling method and lets the user select any sub-expression of this statement for further debugging. If the user manually selects a method call for further debugging, the debugging process can automatically be led into the called method. Nevertheless, inside the called method all output connections have also to be set manually, because no information about correct or incorrect variable occurrences at expression level exists in the calling method. Figure 19.7 shows the GUI, which is used by the `JADE` system to display a fault located in an assignment statement. All sub-expressions of the variable assignment's right-hand side are listed and can be selected by the user. The options left to the user are the following: (1) Stop the current debugging session (`End`), (2) go back to the diagnosis at the outer hierarchy level (`Backtrack`), or (3) debug another method, if a method call is selected in the sub-expression list box (`Debug Method`).

In the following chapters we give some results obtained from experiments with the `JADE` debugging environment. We describe the current `JADE` test suite and

discuss the outcomes of all tests carried out.

Chapter 20

Empirical Results

In this chapter we evaluate the performance of the JADE debugging environment and discuss the advantages and drawbacks of the approaches described in previous chapters of this work. In particular, we test the following properties of the JADE system:

Diagnosis performance: First, we examine the ability of the JADE tool to reduce the search space of bug candidates for a faulty Java method. This is done by creating a system description (see Chapter 15) of the method under consideration and computing all diagnoses as described in Chapter 16. In other words, we test what parts of a Java method can automatically be excluded from the fault localization process in a single diagnosis step and what parts of the search space remain for further debugging actions. Empirical results of the JADE environment and some theoretical considerations about the diagnosis performance are part of this chapter.

Debugging performance: In a second step we evaluate the debugging performance of the JADE debugger (see Chapter 19). This includes empirical results of debugging sessions, which demonstrate in how many steps and with how much user interaction a fault in a Java method is exactly located. The debugging performance builds on the diagnosis performance, but also takes into consideration issues arising in the context of hierarchical debugging, variable queries, and measurement selection (see Chapter 17). The main goal of these tests is to show, how well the JADE debugging environment succeeds in supporting the user during a whole debugging session.

20.1 Evaluating the diagnosis performance

In this section we test the ability of the JADE debugging environment to compute bug candidates for a single successful test-case. In particular, we take a Java method and install a single fault in a way that all pre-conditions for the diagnosis process are still met (see Section 5.2). We then specify a set of test-cases for the given method and run the method on all test-cases. As a consequence, the set of test-cases can be divided into the following two classes: (1) Test-cases,

where an output error can be observed, i.e., positive or successful test-cases, and (2) test-cases, where no misbehavior of the tested method can be identified, i.e., negative or unsuccessful test-cases. Since the installed fault gets observable as a failure or an error only in the case of successful test-cases, these test-cases are used during the diagnosis process. We use a single successful test-case together with its evaluation trace (in case of *ETFDMs*) to compute all diagnoses, i.e., all statements possibly containing the bug. This section contains some theoretical thoughts and empirical results clarifying the performance of our diagnosis approach.

20.1.1 General remarks

As already said in Part II of this work, all used models represent the analyzed `Java` system at statement level. Therefore, in this section we compute diagnoses at statement level, too. To be more precise, we compute all top-level statements of a given method m, which possibly include the installed fault. However, we do not consider statements or expressions nested in sub-blocks, such as branches of selection statements or bodies of loops. In contrast to the debugging process, where we perform a hierarchical fault localization process, here we are only interested in the percentage of all top-level statements, which can be eliminated from the debugging scope by performing a single diagnosis step. Moreover, during all tests we only consider single-diagnoses, i.e., diagnoses containing one statement. As we install single faults into the analyzed methods single-diagnoses exist for all successful test-cases.

Clearly, when computing the set of all single-diagnoses D at statement level for a given block b of length n we expect D to contain at least one possible culprit and in the worst case scenario all n statements of block b. More formally, $1 \leq |D| \leq n$ holds. Ideally, we only get one single-diagnosis, in which case the fault localization process can be terminated after the diagnosis process. However, due to the complex FD structure of most methods we expect $|D|$ to be greater than one in most cases. The following sections give some empirical results and theoretical considerations about the amount of single-diagnoses computed for incorrect `Java` methods.

20.1.2 Empirical diagnosis results

The following paragraphs discuss various test series carried out with different `Java` systems. The source code of all tested systems including the exact specification of all used test-cases is not depicted in this work, but can be downloaded from the `JADE` web-site[1]. In all cases the models described in Part II of this work are used to create system descriptions of the tested methods. All single diagnoses are then computed by setting the maximum number of diagnoses to 100 and the maximum size of diagnoses to 1 (see Section 19.3). The results of all test series are given in Tables 20.2 to 20.7. The meaning of the values stated in each column of these tables is described in Table 20.1. In the following paragraphs, we discuss

[1] http://www.dbai.tuwien.ac.at/proj/Jade/javaPrograms/

#:	Number of test.
Method:	Signature of the tested method.
F:	Number of single fault installed into the method.
TC:	Number of test-case used during the test.
S:	Number of statements of the method.
D_1:	Number of diagnoses computed when using the method's *DFDM*. This number is equal to the number of statements, which are left as possible culprits after the diagnosis process. Note that in most tests the number of diagnoses is the same for *DFDMs* and simplified *DFDMs*. In test cases where this is not the case the number of diagnoses obtained by using the full version of the *DFDM* is given in brackets.
$D_1(\%)$:	Percentage of all statements, which possibly contain the fault after the diagnosis process has been performed using a *DFDM*, i.e., $D_1(\%) = D_1/S$.
D_2:	Number of diagnoses computed when using the method's *ETFDM*. Note that currently the `JADE` system does not support diagnosis using full versions of the *ETFDM*. Therefore, in all cases a simplified *ETFDM* is used.
$D_2(\%)$:	Percentage of all statements, which possibly contain the fault after the diagnosis process has been performed using a *ETFDM*, i.e., $D_2(\%) = D_2/S$.

Table 20.1: Columns of Tables 20.2 to 20.7

the results of each test series in more detail. Finally, we look at some statistics of all performed tests and discuss the implications of the conducted tests.

Test series 1 (*Adder*): The first test series uses an implementation of a binary full adder as the target `Java` system. This implementation includes only method calls and variable assignments, but no selection and loop statements. Furthermore, no explicit object-oriented structures, such as multiple objects, inheritance, or polymorphism are used. Method *adder(int a, int b, int c)* is tested with 8 different single faults and multiple test-cases for each fault, leading to an overall number of 14 tests performed with successful test-cases.

On average 8.14 single-diagnoses are computed. This means that on average 8.14 statements out of a total of 17 have to be considered as possible culprits of the installed single fault, which amounts to 48% of the method's source code. In other words, 8.86 statements can be eliminated from the fault localization process by showing that their behavior does not account for the misbehavior spotted in combination with the used test-case. This amounts to 52% of the method's source code.

Note that the results are exactly the same regardless of the underlying model. This can be explained as follows: Since *adder(int a, int b, int c)* contains no selection or loop statements, there exists only a single control flow path through the method. As a consequence, the method's *DFDM* and *ETFDM* contain exactly the same FDs. Using the simplified version of either the *DFDM* or *ETFDM* does not affect the results either. This is, because no objects are created and no variables of reference type are used in this test series.

The main reason for the high number of statements eliminated in a single diagnosis process is the fact that *adder(int a, int b, int c)* computes the values of two different variables in two independent source code parts. This can also be spotted in the underlying FDMs, which include two independent FD chains. As each single fault only influences one of these FD chains, all statements associated with FDs in the other chain can be eliminated.

#	Method	F	TC	S	D_1	D_1(%)	D_2	D_2(%)
1	*adder(int a, int b, int c)*	f_1	t_1	17	5	29	5	29
2	*adder(int a, int b, int c)*	f_1	t_2	17	5	29	5	29
3	*adder(int a, int b, int c)*	f_2	t_1	17	5	29	5	29
4	*adder(int a, int b, int c)*	f_2	t_2	17	5	29	5	29
5	*adder(int a, int b, int c)*	f_2	t_3	17	5	29	5	29
6	*adder(int a, int b, int c)*	f_3	t_1	17	10	59	10	59
7	*adder(int a, int b, int c)*	f_3	t_2	17	10	59	10	59
8	*adder(int a, int b, int c)*	f_4	t_1	17	10	59	10	59
9	*adder(int a, int b, int c)*	f_4	t_2	17	10	59	10	59
10	*adder(int a, int b, int c)*	f_5	t_1	17	10	59	10	59
11	*adder(int a, int b, int c)*	f_5	t_2	17	10	59	10	59
12	*adder(int a, int b, int c)*	f_6	t_1	17	10	59	10	59
13	*adder(int a, int b, int c)*	f_7	t_1	17	9	53	9	53
14	*adder(int a, int b, int c)*	f_8	t_1	17	10	59	10	59
$\sum$				238	114	48	114	48
∅				17	8.14	48	8.14	48

Table 20.2: Diagnosis results of test series *Adder*

In general this can be explained as follows: Let us look at the digraph of the internal model of a given method's top-level block. Examples of digraphs are depicted in Figures 6.4 and 6.5. We distinguish the following two cases:

1. Assume that all nodes of the digraph are connected with each other either directly or via a finite number of intermediate nodes. With a single (negative) observation at the method's output, we then get all statements associated with one of the variable occurrences in the digraph as bug candidates. By increasing the number of negative observations at the method's output, we also increase the number of conflicts computed during the diagnosis process and thus reduce the number of resulting diagnoses.

2. Assume, on the other hand, that the digraph contains sub-graphs, which are not connected with each other. Then a single fault can only be observable in one of these sub-graphs and the maximum amount of diagnoses equals the number of statements associated with the variable occurrences in this sub-graph. All statements, which are associated with variable occurrences appearing in other sub-graphs only, are eliminated from the debugging scope during the diagnosis process.

Test series 2 (*IfTest*): The second test series evaluates the performance of the `JADE` system in the context of **if** statements. Ten tests with 5 methods and various installed faults and test-cases are performed. The results of test series 2 are given in Table 20.3.

When using *DFDMs* as the underlying models on average 37% of the methods' statements can be eliminated leaving 63% for further debugging. When applying the full *DFDM* instead of its simplified version results can be improved in the case of method *ifTest4(int a, int b, int c)* and *ifTest5(int a, int b, int c)*, because these methods use objects and references. The more complex model structure and distinction between locations and references accounts for this improvement.

#	Method	F	TC	S	D_1	D_1(%)	D_2	D_2(%)
15	*ifTest1(int a, int b)*	f_1	t_1	4	3	75	3	75
16	*ifTest1(int a, int b)*	f_2	t_2	4	3	75	2	50
17	*ifTest2(int a, int b)*	f_1	t_2	3	2	67	2	67
18	*ifTest2(int a, int b)*	f_2	t_1	3	1	33	1	33
19	*ifTest2(int a, int b)*	f_2	t_2	3	1	33	1	33
20	*ifTest3(int a, int b)*	f_1	t_2	3	2	67	2	67
21	*ifTest3(int a, int b)*	f_2	t_2	3	1	33	1	33
22	*ifTest4(int a, int b, int c)*	f_1	t_2	4	3 (2)	75 (50)	3	75
23	*ifTest4(int a, int b, int c)*	f_2	t_1	4	3 (2)	75 (50)	2	50
24	*ifTest5(int a, int b, int c)*	f_1	t_1	4	3 (2)	75 (50)	3	75
$\sum$				35	22 (19)	63 (54)	20	57
$\varnothing$				3.5	2.2 (1.9)	63 (54)	2.0	57

Table 20.3: Diagnosis results of test series *IfTest*

In case of *ETFDMs* results can also be improved to a total of 43% of all statements being removed by the diagnosis component leaving only 57% for further debugging. The advantages of *ETFDMs* in the context of selection statements are discussed in Section 10.3 in detail and can shortly be summarized as follows:

- Whereas in the computation of the selection statement's FDM *DFDMs* make use of all summarized branch models, *ETFDMs* only take into consideration the summarized model of the branch executed at run-time. Therefore, *ETFDMs* only incorporate FDs produced in one branch, which leads to a smaller amount of FDs in the FDM of the whole selection statement.

- In case of the *DFDM* additional self dependencies are introduced in certain cases (see Section 10.3.2). This is not necessary in the case of *ETFDMs*, leading, again, to fewer FDs.

Note that similar results can be expected for **switch** statements and conditional expressions of the form $expr_1 \,?\, expr_2 \,:\, expr_3$. Further tests with the `JADE` systems will have to verify that.

Test series 3 (*WhileTest*): The next test series deals with the evaluation of methods containing **while** statements. 10 tests are carried out with two methods and various fault/test-case pairs. The results of all tests are depicted in Table 20.4.

The results are similar to test series *IfTest*. In particular, when using *DFDMs* 41% of the methods' statements can be eliminated leaving 59% for further debugging. Again, *ETFDMs* outperform *DFDMs* by eliminating 53% of all statements leaving only 47% for further analysis. The main advantages of *ETFDMs* in comparison to *DFDMs* in the context of the modeling of loop statements are discussed in Section 10.4 in detail. They can be summarized as follows:

- *ETFDMs* use different models of a loop statement's body for different iterations. *DFDMs* on the other hand incorporate the FDs from all iterations

#	Method	F	TC	S	D_1	D_1(%)	D_2	D_2(%)
25	*whileTest1(int i)*	f_1	t_4	7	6	86	3	43
26	*whileTest1(int i)*	f_2	t_2	7	6	86	3	43
27	*whileTest1(int i)*	f_3	t_4	7	5	71	3	43
28	*whileTest2(int i)*	f_1	t_1	5	2	40	1	20
29	*whileTest2(int i)*	f_1	t_2	5	2	40	2	40
30	*whileTest2(int i)*	f_1	t_3	5	2	40	2	40
31	*whileTest2(int i)*	f_2	t_2	5	2	40	×(5)	×(100)
32	*whileTest2(int i)*	f_2	t_3	5	2	40	2	40
33	*whileTest2(int i)*	f_3	t_2	5	3	60	2	40
34	*whileTest2(int i)*	f_3	t_3	5	3	60	2	40
$\sum$				56	33	59	25	47
∅				5.6	3.3	59	2.5	47

Table 20.4: Diagnosis results of test series *WhileTest*

into a single model. The latter model contains more FDs than the individual models used by the *ETFDM* modeling component, if different iterations produce different body-models, e.g., in the case of selection statements nested in the loop body.

- *DFDMs* accumulate the summarized body-models of all iterations, which leads to a monotonic increasing amount of FDs at the loop's top-level. *ETFDMs* on the other hand combine the individual summarized body-models in such a way that FDs, which are valid only for previous iterations, are eliminated in the currently modeled iteration (see Section 10.4).

Since no methods used in this test series make use of objects or reference variables, the use of full models does not change the results in Table 20.4. Note that similar results can be expected for methods including **do** and **for** statements, because of the similar modeling algorithms.

The computation of all single-diagnoses for method *whileTest2(int i)* with fault f_2 and test-case t_2, i.e., test 31, with an *ETFDM* highlights an interesting drawback of *ETFDMs* in comparison to *DFDMs*. Here no diagnoses are computed by the `JADE` system, although $t2$ represents a successful test-case. The reason for this behavior and possible approaches to deal with this type of problems are discussed in Section 21.1.2. Since no diagnoses are eliminated due to this problem, we use the total amount of statements in *whileTest2(int i)*, i.e., 5, instead of 0 for further statistical computations. Clearly, in this case 100% of the source code remain for further debugging.

Test series 4 (*Numeric*): Another test series is carried out with two numeric test methods computing the difference quotient and integral of a given method. Table 20.5 shows all results of test series *Numeric*.

Method *differentiate(double x, double diff)* has only one possible control flow path and a single FD chain in both model types. Therefore, in this case no diagnoses can be eliminated using either the *DFDM* or the *ETFDM*. As a consequence 100% of the source code remain as bug candidates.

The same is true for method *integrate(double a, double b, int n)*, although this method includes a **while** statement. Note that in tests 39, 40, and 41 no

#	Method	F	TC	S	D_1	D_1(%)	D_2	D_2(%)
35	*differentiate(double x, double diff)*	f_1	t_2	4	4	100	4	100
36	*differentiate(double x, double diff)*	f_1	t_3	4	4	100	4	100
37	*differentiate(double x, double diff)*	f_2	t_2	4	4	100	4	100
38	*differentiate(double x, double diff)*	f_2	t_3	4	4	100	4	100
39	*integrate(double a, double b, int n)*	f_1	t_1	5	5	100	×(5)	×(100)
40	*integrate(double a, double b, int n)*	f_1	t_2	5	5	100	×(5)	×(100)
41	*integrate(double a, double b, int n)*	f_1	t_3	5	5	100	×(5)	×(100)
42	*integrate(double a, double b, int n)*	f_2	t_1	5	5	100	5	100
43	*integrate(double a, double b, int n)*	f_2	t_3	5	5	100	5	100
$\sum$				41	41	100	41	100
∅				4.6	4.6	100	4.6	100

Table 20.5: Diagnosis results of of series *Numeric*

#	Method	F	TC	S	D_1	D_1(%)	D_2	D_2(%)
44	*trafficLightTest(int i)*	f_1	t_5	5	3	60	3	60
45	*trafficLightTest(int i)*	f_2	t_2	5	3	60	3	60
46	*trafficLightTest(int i)*	f_2	t_3	5	3	60	3	60
47	*trafficLightTest(int i)*	f_2	t_4	5	3	60	3	60
$\sum$				20	12	60	12	60
∅				5	3	60	3	60

Table 20.6: Diagnosis results of test series *TrafficLight*

diagnoses can be computed, if the *ETFDM* is used as underlying model. The reasons for this behavior are presented in Section 21.1.2.

Test series 5 (*Trafficlight*): Test series *Trafficlight* implements a small traffic light simulation including objects, loops, and selection statements. The results of all tests carried out with method *trafficLightTest(int i)* are presented in Table 20.6.

In all test-cases 40% of the statements can be eliminated as bug candidates leaving 60% of the source code as possible bug locations. Interestingly, no improvement can be obtained by using *ETFDMs*, despite the appearance of loops and **if** statements in the tested method's body. This can be explained as follows: (1) The loop body is executed at least once in all test-cases. This is, because no successful test-case can be found, in which the loop body is never executed. As a consequence, the loop statement cannot be eliminated from the bug candidates in any of the test-cases. (2) The nested **if** statements all produce the same FDs. Therefore, the *ETFDM* cannot optimize the resulting model.

Using the full *DFDM* instead of its simplified version also results in no improvements. This is due to the fact that all statements of the tested method depend on the same instance field of the created traffic light object. Keeping this field separate from other locations and references does not improve diagnosis.

Test series 6 (*Library*): The last test series implements a fully object-oriented virtual library. Authors and books are created and stored in a library object by the use of a linked list. The tested method then computes the number

#	Method	F	TC	S	D_1	D_1(%)	D_2	D_2(%)
48	*library()*	f_1	t_1	26	21 (19)	81 (73)	21	81
49	*library()*	f_2	t_1	26	20 (18)	77 (69)	18	69
50	*library()*	f_3	t_1	26	21 (18)	81 (69)	21	81
51	*library()*	f_4	t_1	26	20 (17)	77 (65)	20	77
52	*library()*	f_5	t_1	26	21 (18)	81 (69)	20	77
$\sum$				130	103 (90)	79 (69)	100	77
$\varnothing$				26	20.6 (18)	79 (69)	20	77

Table 20.7: Diagnosis results of test series *Library*

#:	Number of test series.
Test series:	Name of test series.
#TC:	Total number of test-cases in test series.
$\varnothing S$:	Average number of statements of the methods in the test series.
$\varnothing D_1$:	Average number of diagnoses when using *DFDMs* of all methods in the test series. Values in brackets stand for results obtained from tests with full models instead of simplified ones. These values are only specified, if different from the ones obtained by using simplified models.
$\varnothing D_1$(%):	$\varnothing D_1(\%) = \varnothing D_1 / \varnothing S$.
$\varnothing D_2$:	Average number of diagnoses when using simplified *ETFDMs* of all methods in the test series.
$\varnothing D_2$(%):	$\varnothing D_2(\%) = \varnothing D_2 / \varnothing S$.

Table 20.8: Columns of Table 20.9

of books published by the author, who has published most books out of all authors, whose books are currently in the library. The results of all tests performed with method *library()* are depicted in Table 20.7.

When using a *DFDM* as the underlying model only 21% of the source code can be eliminated, because of the rather complex object structure in method *library()*. This complex structure is due to a high number of created objects, which are all referenced by each other mainly by the use of a linked list.

If the full *DFDM* is used instead of its simplified version, the diagnosis performance can be improved quite notably to 31% of all statements being removed from the debugging focus. In this case this is clearly due to the more detailed representation of the complex object structure. Statements influencing only instance fields, which are not used in the computation of the incorrect output values can be eliminated, which is not the case, if object structures are used instead of locations and references (see Chapter 14).

The use of an *ETFDM* shows a slight improvement in comparison with the *DFDM*, which is probably due to the **while** loop in method *library()*. In this test series the combination of full models separating locations from references and the advantages of *ETFDMs* looks especially promising. Further tests with future versions of the `JADE` system could verify this assumption.

20.1.3 Discussion

Table 20.9 summarizes the average diagnosis results of all test series carried out. Table 20.8 describes the meaning of the columns of Table 20.9. The main findings of the performed tests can be summarized as follows:

- If a *DFDM* is used to compute all single-diagnoses of a given `Java` method,

#	Test series	#TC	$\varnothing S$	$\varnothing D_1$	$\varnothing D_1(\%)$	$\varnothing D_2$	$\varnothing D_2(\%)$
1	*Adder*	14	17	8.14	48	8.14	48
2	*IfTest*	10	3.5	2.2 (1.9)	63 (54)	2.0	57
3	*WhileTest*	10	5.6	3.3	59	2.5	47
4	*Numeric*	9	4.6	4.6	100	4.6	100
5	*Trafficlight*	4	5	3	60	3	60
6	*Library*	5	26	20.6 (18)	79 (69)	20	77
$\sum$		52	520	325.4 (309.4)		312.4	
$\varnothing$		1	10	6.3 (5.9)	63 (59)	6	60

Table 20.9: Average diagnosis results of all test series

the amount of statements possibly including a bug can be reduced in most cases. Therefore, the search space for additional debugging steps can be limited quite efficiently. The tests described in this section indicate that approximately 37% of a method's statements can be eliminated leaving 63% as possible culprits of a given bug. Whereas in some cases more than 50% of the source code can be eliminated due to multiple independent FD chains (see test series *Adder*), in other cases all statements remain in the debugging focus, because of a complex FD structure, where all statements lie in the same FD chain (see test series *Numeric*). It can be expected that larger methods, i.e., methods with more statements in their top-level block, intensify this behavior by either eliminating large parts of their source code or resulting in nearly all their statements as bug candidates. However, it has to be noted that the diagnosis performance of the `JADE` debugger on its own is not sufficient to exactly localize source code bugs at a method's top-level.

- If *ETFDMs* are used instead of *DFDMs* results can be improved slightly. In particular, with *ETFDMs* 40% of a method's statements can be eliminated leaving only 60% of the source code as possible culprits. The improvement is mainly due to the more exact modeling of selection and loop statements, which results in fewer FDs and thus less diagnoses being created. Although *ETFDMs* improve the results quite significantly in some test series, this approach is still not sufficient for an exact fault localization process.

- If *DFDMs* are used in their full versions instead of their simplified versions, a further improvement is obtained in cases, where multiple objects and references are used. This is mainly due to the more detailed representation of the full *DFDM*, which keeps locations and variables of reference type separate (see Chapter 8). In particular, in all tests an average of 41% instead of 37% of all statements can be removed from the debugging focus. Unfortunately, the `JADE` debugging environment currently does not support the use of full *ETFDMs*. In future releases such a feature promises another improvement by combining the advantages of *ETFDMs* and full models. This combination seems possible, because both techniques constitute independent approaches.

Note that the used test methods are all small to medium-size Java methods. This has the following two reasons: (1) Most methods are designed to explicitly test and demonstrate certain modeling and diagnosis features of the JADE system, e.g., selection statements, loops, etc... (2) Larger applications, which include more statements at their top-levels can hardly be found, since most methods contain only a few loop or selection statements at their top-level block. The main difference is that larger methods contain more nested blocks, which can only be tested in the context of hierarchical debugging during concrete debugging sessions. Nevertheless, additional test series should be performed in the future to further evaluate and improve the diagnosis ability of the JADE debugger. The focus of these test series should be on one of the following issues:

- Tests with longer and more sophisticated methods. This also includes different source code structures and new classes of installed faults.
- Tests with more observations. So far only observations for output connections of the system descriptions have been specified. Additional observations, possibly specified by the user through the use of assertions, are expected to improve diagnosis results.
- Tests with methods, in which multiple faults are installed. These methods would then have to be diagnosed by allowing the diagnosis engine to compute larger diagnoses.
- Tests with the concurrent application of multiple test-cases (see Section 18.2).

As shown in this section the application of MBD techniques to compute bug candidates for buggy Java systems cannot only be put on a firm theoretical basis, but also provides a means of notably reducing the amount of statements possibly including a given fault. Nevertheless, it is obvious from the results presented in this section that the computation of diagnoses on its own cannot limit the search space in such a way that exact bug locations can be determined automatically. Furthermore, a single diagnosis step does not provide sufficient information for the user to non-ambiguously identify a bug location in an interactive procedure. Therefore, the computation of diagnoses can only be seen as the base technique for two more general approaches: (1) The computation of diagnoses is one element of an iterative and interactive debugging environment as described in Chapters 17 and 19. The following sections evaluate the performance of the JADE debugger over the whole debugging process and show how efficient it can support a user during a complete fault localization process. (2) The diagnosis process itself can be lifted to a more abstract level in the course of a debugging process. This means that building on modeling techniques described in this work models for whole methods or even modules can be created and used to locate bugs at these higher levels in a system architecture. The exact fault localization at statement or expression level can then be done by using more detailed models, such as value-based models. Section 21.2.2 includes a more detailed discussion about this possible use of FDMs.

20.2 Evaluating the debugging performance

In contrast to the previous section this section is dedicated to the evaluation of the performance of the JADE debugging environment during the whole debugging process, which is depicted in Figure 17.1. As a consequence, we are interested in the debugger's ability to support a user during the whole debugging process until eventually the exact location of a given bug is found. In particular, this no longer includes only the diagnosis process, but also an evaluation of the used measurement selection algorithm, variable queries, and support for the handling of hierarchical debugging steps, for example, in the case of selection and loop statements.

20.2.1 General remarks

In the course of a debugging session the programmer normally brings in a lot of a priori knowledge about the type and location of a certain bug. Apart from knowledge about the intended data and control flow of the method in question, he might have some additional information. Examples of such an information are the programmer's experience with fault probabilities of various statements and expressions or intuitive feelings about the correctness of certain parts of his work. In order to test the JADE debugger it is assumed that no knowledge about the type and the location of the fault is available apart from the specification of the method's in/out behavior and its internal data and control flow. One way to incorporate a priori knowledge into the debugging process is the use of assertions, which is discussed in Section 18.1. Other approaches, such as fault probabilities of statements and expressions or the definition of different fault modes could be introduced into future versions of the JADE system.

In analogy to the evaluation of the system's diagnosis performance, we further assume that at the moment only test methods with single faults are considered and during the diagnosis steps only single-diagnoses are computed. As before, debugging is performed only at statement level. In contrast to the diagnosis process, the exact location of a given bug is detected not only in a method's top-level block, but also in sub-blocks until a fault location has non-ambiguously been identified. The debugging of expressions (mainly to identify faults in method calls in order to restart the automatic process in the called method) has to be conducted manually as described in Section 19.4. Applying model-based techniques to the debugging at expression level might be implemented in future versions of the JADE debugger.

Ideally, an evaluation of a debugging tool should be done by an empirical experiment, which involves the formation of two groups of programmers. Each group includes programmers from different backgrounds and with different programming experiences. The overall skills and experiences of all participants of the experiment have to be distributed evenly across both groups in order to allow for comparisons between the two groups. The members of both groups are then confronted with Java systems exhibiting incorrect behavior. Note that in such a way no a priori knowledge about the possible locations and natures of the individual faults are available to the programmers. Whereas the members

of one group then have to locate the faults in all test methods by the use of traditional debugging tools, the participants in the other group can make use of the JADE debugging tool. If we assume that both groups are equally familiar with their respective tool and that both tools have comparable learning times, the average times needed by both groups for exactly the same fault localization tasks would provide a good measurement of the performance of the JADE tool in relation to existing standard debugging environments. However, this approach is a very complex and time-consuming task, which not only relies on a stable and complete version of the JADE debugger, but also takes up a lot of resources, especially as far as human work force and organizational constraints are considered. Therefore, in this section we make use of a simplified evaluation process of the JADE debugging environment and leave the approach described above to future research.

The approach pursued in this work contrasts the JADE debugging environment with traditional debugging tools by comparing the amount of user interactions needed by both systems to exactly locate a single fault in a test method. If we look at traditional debuggers we find that, without any a priori knowledge, we have to go through the source code of a method statement by statement. This means that, if a bug occurs at statement line i of a certain block, the programmer has to look at all i statements in order to find the bug. In reality an experienced programmer might know, whether a statement might account for an observed failure or not, but in an object-oriented environment with a lot of side-effects such an assessment can become a hard task. Together with the abovementioned assumption about a priori knowledge, we assume that i user interactions, i.e., evaluations of a certain statement and its corresponding variable environment, are needed to find a bug in line i of a certain method. In the best case we find the bug in only one step, but on the other hand the worst case scenario is going through all statements of a method to locate a bug. Given a concrete test-case, we assume that the index of the buggy statement within a method determines the number of user interactions needed to locate the bug. This number is the reference value for the performance of the JADE debugger. More formally, we define:

Definition 20.2.1 *The reference value R for the performance of the JADE debugger is defined as the index of the statement containing the installed source code bug.*

In the current version of the JADE debugging environment various kinds of user interactions are performed (see Section 19.3). The following list describes all user interactions, which are used by the JADE debugging tool, and highlights the problems of comparing these interactions with the reference value R:

Variable setup: Before a certain method can automatically be debugged, the user has to evaluate the method's output behavior by specifying a negative observation for at least one of the system's output connections. These interactions are essential to the debugging process, because without them no diagnoses can be computed. However, in a well-defined software engineering process these observations can directly be taken from the test phase as

the target values of the used test-cases. From the latter point of view the variable setup is no longer a user interaction as far as the debugging process is concerned, because the needed information can be taken from earlier stages in the software development process.

Variable queries: These user interactions are needed to find out, whether the value of a particular variable at a certain source code position is correct or not. Variable query interactions are similar to the evaluation of a statement in a traditional debugging process and therefore constitute the main type of user interaction, which determines the performance of the `JADE` debugging tool. The problem with comparing variable queries with the evaluation of statements in traditional debuggers is that a single statement normally contains multiple variable occurrences. It is therefore possible to get multiple variable queries aiming at different variable occurrences within the same statement during a single debugging process.

Selection statement interactions: These interactions are designed to find out, whether the condition of a faulty selection statement is correct or not. In case it is, the debugging process is automatically led into the branch of the selection statement executed at run-time. Otherwise, the fault is located in the condition. Strictly speaking, these interactions belong to the debugging at expression level and are therefore not covered by the reference value R.

Loop interactions: These interactions let the user specify (1) whether a certain fault is located in the body or the condition of an incorrect loop statement and (2) the first iteration, in which the fault becomes observable as a failure or an error. Again, these tasks represent debugging actions at expression level and are not unique to the `JADE` debugger. It could therefore be assumed that traditional debuggers use the same technique or have to go through all iterations, what in general leads to an even worse performance. In both cases these interactions are not covered by the reference value R.

Method call interactions: These interactions are designed to find the smallest sub-expression of a faulty statement, which contains the bug. In case of buggy method and constructor calls further debugging is automatically performed inside the called method. Again, these interactions represent debugging at expression level and are thus not covered by the reference value R.

When testing the performance of the `JADE` debugger we can now specify the amount of user interactions for each of the described types of interactions. In order to compare it with the reference value R provided by the performance of traditional debuggers we compute two indices. First, all interactions are aggregated. The second index only takes variable query interactions into account, because, as discussed above, only these interactions represent interactions at statement level. More formally, we define:

#:	Number of performed test.
S:	Number of statements of the method.
R:	Index of the statement containing the fault, i.e., reference value for performance of the JADE debugger.
B:	Number of setup interactions specifying the output values of the tested method.
Q:	Number of variable queries.
I:	Number of selection statement interactions.
W:	Number of loop interactions.
T:	Total number of user interactions.

Table 20.10: Columns of Tables 20.11 to 20.16

Definition 20.2.2 *The performance of the JADE debugging environment is measured by the following two indices:*

- *Q is defined as the total number of variable queries during a single debugging session.*
- *T is defined as the total number of user interactions during a single debugging session.*

From what has been said about the individual types of user interactions it should become clear that there exist a couple of difficulties in the comparison of both indices with the reference value *R*. However, such a comparison can still provide meaningful information about the performance of the JADE tool, especially in relation to standard debugging environments. In the following section we present some empirical results obtained from experiments with the JADE debugger and Java test methods.

20.2.2 Empirical debugging results

In this section we test the debugging potential of both models, the *DFDM* and *ETFDM*, on the same test methods, which are used in Section 20.2.2 during the evaluation of the diagnosis performance of the JADE system. The results of all 6 test series are given in Figures 20.11 to 20.16, whose columns are described in detail in Table 20.10. The following paragraphs discuss the results of the individual test series in more detail.

Note that all tests are carried out with simplified versions of the *DFDMs* and *ETFDMs* of all test methods. Full models are not tested due to the following reasons: (1) The handling of full models during the debugging process is very complex, especially as far as the specification of observations is concerned. This problem and possible enhancements are discussed in Section 21.2.1 in more detail. (2) As already mentioned the current version of the JADE debugger does not support the creation of full versions of *ETFDMs*. Therefore, a comparison of the full versions of both model types is not possible. In all tests the source code fault can be found directly in the tested method. Therefore, method call interactions are not needed and thus not stated in the following paragraphs.

Test series 1 (*Adder*): Table 20.11 shows the results obtained from test series *Adder*. Whereas on average 10 statements have to be checked to assuredly

#	S	R	*DFDM*					*ETFDM*				
			B	Q	I	W	T	B	Q	I	W	T
1	17	4	1	2	0	0	3	1	2	0	0	3
2	17	4	1	2	0	0	3	1	2	0	0	3
3	17	7	1	2	0	0	3	1	2	0	0	3
4	17	7	1	2	0	0	3	1	2	0	0	3
5	17	7	1	2	0	0	3	1	2	0	0	3
6	17	12	1	3	0	0	4	1	3	0	0	4
7	17	12	1	3	0	0	4	1	3	0	0	4
8	17	11	1	3	0	0	4	1	3	0	0	4
9	17	11	1	4	0	0	5	1	4	0	0	5
10	17	14	1	4	0	0	5	1	4	0	0	5
11	17	14	1	4	0	0	5	1	4	0	0	5
12	17	16	1	3	0	0	4	1	3	0	0	4
13	17	12	1	3	0	0	4	1	3	0	0	4
14	17	9	1	3	0	0	4	1	3	0	0	4
$\sum$	238	140	14	40	0	0	54	14	40	0	0	54
$\varnothing$	17	10	1	2.9	0	0	3.9	1	2.9	0	0	3.9

Table 20.11: Debugging results of test series *Adder*

find the bug with a traditional debugging approach, the `JADE` debugging environment performs less than 4 user interactions and less than 3 variable queries. Since no selection and loop statements are used during this first test series, no hierarchical debugging is performed and only variable setup and variable query interactions are carried out. As already described in Section 20.1.2 quite a large part of method *adder(int a, int b, int c)* can be excluded from the debugging process during the first diagnosis process. After that the repeated application of measurement selection steps, variable queries, and new computations of diagnoses leads to the exact localization of the installed fault. Note that similar to the computation of diagnoses there are no differences between the two tested models. The debugging procedures for the *DFDM* and *ETFDM* are exactly the same, because no loop and selection statements are used in this test series.

Test series 2 (*IfTest*): Table 20.12 shows all results obtained from experiments during test series *IfTest*. On average the fault is installed in statement line 4.9 and can be located with the `JADE` tool with 3 interactions (in case of the *DFDM*) or 2.8 interactions (in case of the *ETFDM*). If we look at the number of variable queries, only 1.4 (*DFDM*) and 1.2 (*ETFDM*) user interactions are needed. In contrast to test series *Adder* selection statement interactions are performed in cases where the bug can be found in a statement nested in the branch of an **if** statement. Note that in test 20 two **if** interactions are needed to located a fault in the branch of an **if** statement, which is itself nested in another **if** statement.

If an *ETFDM* is used instead of the *DFDM* in two cases less variable queries are performed, which is due to the more efficient diagnosis process in these cases (see Section 20.1.2). Note that with both models the same number of selection statement interactions is performed, since the hierarchical debugging is not influenced by the different modeling approaches.

#	S	R	*DFDM*					*ETFDM*				
			B	Q	I	W	T	B	Q	I	W	T
15	5	3	1	2	1	0	4	1	2	1	0	4
16	5	2	1	2	0	0	3	1	1	0	0	2
17	5	4	1	1	1	0	3	1	1	1	0	3
18	5	5	1	0	0	0	1	1	0	0	0	1
19	5	5	1	0	0	0	1	1	0	0	0	1
20	9	7	1	1	2	0	4	1	1	2	0	4
21	9	9	1	0	0	0	1	1	0	0	0	1
22	6	5	1	3	1	0	5	1	3	1	0	5
23	6	2	1	2	0	0	3	1	1	0	0	2
24	8	7	1	3	1	0	5	1	3	1	0	5
$\sum$	63	49	10	14	6	0	30	10	12	6	0	28
∅	6.3	4.9	1	1.4	0.6	0	3	1	1.2	0.6	0	2.8

Table 20.12: Debugging results of test series *IfTest*

#	S	R	*DFDM*					*ETFDM*				
			B	Q	I	W	T	B	Q	I	W	T
25	11	5	1	3	0	0	4	1	2	0	0	3
26	11	3	1	5	0	0	6	1	3	0	0	4
27	11	9	2	3	0	1	6	2	2	0	1	5
28	12	3	1	3	0	0	4	1	0	0	0	1
29	12	3	1	2	0	0	3	1	2	0	0	3
30	12	3	1	3	0	0	4	1	1	0	0	2
31	12	5	1	1	0	1	3	×	×	×	×	×
32	12	5	1	2	0	1	4	1	2	0	1	4
33	12	9	2	4	0	2	8	2	2	0	2	6
34	12	9	3	4	0	2	9	3	2	0	2	7
$\sum$	117	54	14	30	0	7	51	13	16	0	6	35
∅	11.7	5.4	1.4	3	0	0.7	5.1	1.3	1.6	0	0.6	3.5

Table 20.13: Debugging results of test series *WhileTest*

Test series 3 (*WhileTest*): The third test series tests the JADE debugger's performance on methods including **while** statements. Table 20.13 shows all results of this test series. On average the bug can be found in statement line 5.4 and it takes 5.1 user interactions (in case of the *DFDM*) or 3.5 user interactions (in case of the *ETFDM*) to exactly localize this fault. Looking only at variable queries, 3 such interactions are needed with the *DFDM* and only 1.6 in case of the *ETFDM*. Due to the reasons explained in Section 20.1.2 the results are significantly better, when *ETFDMs* are used as the underlying models. However, in test 31 the debugging process is terminated after the initial diagnosis step, because no diagnoses can be computed (see Section 20.1.2). This drawback of *ETFDMs* is discussed in more detail in Section 21.1.2.

Test series 4 (*Numeric*): The results of test series *Numeric* are given in Table 20.14. Here, more interactions are needed than in a manual walk-through. In particular, the fault can be found in statement 3.6 on average and localized only after 4.4 (*DFDM*) or 5.3 (*ETFDM*) user interactions, respectively. The number of variable queries is also high with 3 in case of *DFDMs* and 3.7 in case of *ETFDMs*. The main reason for this poor performance is the fact that in all

#	S	R	*DFDM*					*ETFDM*				
			B	Q	I	W	T	B	Q	I	W	T
35	4	3	1	1	0	0	2	1	2	0	0	3
36	4	3	1	2	0	0	3	1	2	0	0	3
37	4	4	1	3	0	0	4	1	3	0	0	4
38	4	4	1	3	0	0	4	1	3	0	0	4
39	8	2	1	2	0	0	3	×	×	×	×	×
40	8	2	1	2	0	0	3	×	×	×	×	×
41	8	2	1	2	0	0	3	×	×	×	×	×
42	8	6	2	6	0	1	9	2	6	0	1	9
43	8	6	2	6	0	1	9	2	6	0	1	9
$\sum$	56	32	11	27	0	2	40	8	22	0	2	32
∅	6.2	3.6	1.2	3	0	0.2	4.4	1.3	3.7	0	0.3	5.3

Table 20.14: Debugging results of test series *Numeric*

#	S	R	*DFDM*					*ETFDM*				
			B	Q	I	W	T	B	Q	I	W	T
44	14	11	1	2	3	1	7	1	2	3	1	7
45	14	6	2	2	1	1	6	2	2	1	1	6
46	14	6	2	2	1	1	6	2	2	1	1	6
47	14	6	2	2	1	1	6	2	2	1	1	6
$\sum$	56	29	7	8	6	4	25	7	8	6	4	25
∅	14	7.25	1.75	2	1.5	1	6.25	1.75	2	1.5	1	6.25

Table 20.15: Debugging results of test series *Trafficlight*

test methods of this test series all statements remain as potential culprits after the initial diagnosis process (see Section 20.1.2). Another problem arising in the context of *ETFDMs* is that in tests 39 to 41 the fault cannot be located due to the problem of the *ETFDM* discussed in Section 21.1.2.

Test series 5 (*Trafficlight*): The results of test series *Trafficlight* are depicted in Table 20.15. Whereas on average the fault can be found in statement 7.25, the `JADE` debugger takes 6.25 user interactions, but only 2 variable queries to exactly locate the fault. In this test series both models, the *DFDM* and *ETFDM* perform equally well for reasons explained in Section 20.1.2. A high percentage of all user interactions are selection and loop interactions, which are executed in the course of hierarchical debugging and cannot be optimized by the use of *ETFDMs*. The complicated structure of nested loops and selection statements accounts for the rather poor performance in this test series as far as index T is concerned.

Test series 6 (*Library*): The last test series evaluates the debugger's performance in the context of a real object-oriented library example. All results are depicted in Table 20.16. On average a fault is installed in statement 18.6 and can be located with less than 8 user interactions with both models. The number of variable queries looks even more promising with only 5.6 (*DFDM*) or 5.4 (*ETFDM*) such interactions. Here a higher number of variable assignments and method call statements in relation to loop and selection statements

#	S	R	*DFDM*					*ETFDM*				
			B	Q	I	W	T	B	Q	I	W	T
48	33	30	2	7	1	1	11	2	6	1	1	10
49	33	6	1	5	0	0	6	1	5	0	0	6
50	33	10	1	5	0	0	6	1	5	0	0	6
51	33	16	1	5	0	0	6	1	4	0	0	5
52	33	31	2	6	1	1	10	2	7	1	1	11
$\sum$	165	93	7	28	2	2	39	7	27	2	2	38
$\varnothing$	33	18.6	1.4	5.6	0.4	0.4	7.8	1.4	5.4	0.4	0.4	7.6

Table 20.16: Debugging results of test series *Library*

#:	Number of test series.
Test series:	Name of test series.
#TC:	Total number of test-cases in test series.
$\varnothing S$:	Average number of statements of the methods in the test series.
$\varnothing R$:	Average index of buggy statement.
$\varnothing T_1$:	Average number of total user interactions using the *DFDM*.
$\varnothing T_1(\%)$:	$\varnothing T_1(\%) = \varnothing T_1/\varnothing R$.
$\varnothing T_2$:	Average number of total user interactions using the *ETFDM*.
$\varnothing T_2(\%)$:	$\varnothing T_2(\%) = \varnothing T_2/\varnothing R$.
$\varnothing Q_1$:	Average number of variable query user interactions using the *DFDM*.
$\varnothing Q_1(\%)$:	$\varnothing Q_1(\%) = \varnothing Q_1/\varnothing R$.
$\varnothing Q_2$:	Average number of variable query user interactions using the *ETFDM*.
$\varnothing Q_2(\%)$:	$\varnothing Q_2(\%) = \varnothing Q_2/\varnothing R$.

Table 20.17: Columns of Tables 20.18 and 20.19

leads to significantly better result than in the last two test series. Note that the *ETFDMs* perform slightly better, which is due to the modeling of the **while** and **if** statement in method *library()*.

20.2.3 Discussion

To test the debugging performance of the `JADE` debugging environment, several debugging sessions were carried out with all 52 test methods used in Section 20.1. Tables 20.18 and 20.19 summarize the average debugging results of all test series with respect to the two indices T and Q, respectively. Table 20.17 describes the meaning of the columns of both tables. If we compare the two indices T and Q with the reference value R as defined in Section 20.2.1, the main findings can be summarized as follows:

- In most cases the `JADE` debugging tool needs significantly less user interactions than a traditional debugger, if the latter is used to step through the code statement by statement. In particular, with the *DFDM* only 60 percent of the user interactions of a traditional debugger are needed to exactly locate an installed source code fault. If only variable query interactions are counted only 37% of the user interactions are needed. However, it should be mentioned that there arise various difficulties when comparing these figures as discussed in Section 20.2.1.

- When an *ETFDM* is used instead of a *DFDM* the debugger's performance

#	Test series	#TC	$\varnothing S$	$\varnothing R$	$\varnothing T_1$	$\varnothing T_1(\%)$	$\varnothing T_2$	$\varnothing T_2(\%)$
1	*Adder*	14	17	10	3.9	39	3.9	39
2	*IfTest*	10	6.3	4.9	3	61	2.8	57
3	*WhileTest*	10	11.7	5.4	5.1	94	3.9	72
4	*Numeric*	9	6.2	3.6	4.4	120	5.3	147
5	*Trafficlight*	4	14	7.25	6.25	86	6.25	86
6	*Library*	5	33	18.6	7.8	42	7.6	41
$\sum$		52	694.8	397.4	239.2		232.3	
$\varnothing$		1	13.4	7.6	4.6	60	4.5	59

Table 20.18: Average total user interactions of all test series

can slightly be improved due to a more powerful diagnosis process. This means that the debugging process is exactly the same as far as hierarchical debugging steps (selection statement and loop interactions) are concerned, but less variable queries are needed, because of a more efficient elimination of wrong diagnoses in the individual diagnosis steps. Overall, only 35% of the user interactions of the reference value are needed, if only variable queries are counted.

- Whereas the fault can be located in all cases, if *DFDMs* are used, this is not the case with *ETFDMs*. In particular, in 4 out of the 52 performed tests the debugging process terminates, because no diagnoses can be produced with the *ETFDM*. The reasons for this behavior and possible solutions are discussed in Section 21.1.2.

- In some cases the debugging process of the `JADE` debugger works very efficiently with less than 50% of the user interactions of a traditional debugger. However, in other cases significantly more user interactions are needed with the `JADE` tool. As already mentioned with the `JADE` debugger a single user interaction evaluates a single variable occurrence, whereas in our reference debugger one interaction evaluates the correctness of a whole statement. Therefore, the worst case scenario in the case of the `JADE` debugger involves a much higher number of user interactions, since the number of variable occurrences is normally much higher than the number of statements.

- The crucial element in the debugging process is the diagnosis step, which is evaluated in detail in Section 20.1. Whereas initial variable setup interactions and loop and selection statement interactions for hierarchical debugging steps are inevitable in most cases (and have to be carried out with traditional debugging tools, too), the number of variable queries can be optimized by the use of different models and an efficient measurement selection algorithm.

The last point suggests that the debugging process can mainly be improved by enhancing the diagnosis process and making use of more efficient measurement selection algorithms. The following list shows some further tests, which should be carried out with future versions of the `JADE` tool in order to increase the overall performance during the whole debugging process:

#	Test series	#TC	$\varnothing S$	$\varnothing R$	$\varnothing Q_1$	$\varnothing Q_1(\%)$	$\varnothing Q_2$	$\varnothing Q_2(\%)$
1	*Adder*	14	17	10	2.9	29	2.9	29
2	*IfTest*	10	6.3	4.9	1.4	29	1.2	24
3	*WhileTest*	10	11.7	5.4	3	56	1.8	33
4	*Numeric*	9	6.2	3.6	3	83	3.7	103
5	*Trafficlight*	4	14	7.25	2	28	2	28
6	*Library*	5	33	18.6	5.6	30	5.4	29
$\sum$		52	694.8	397.4	147.6		138.9	
$\varnothing$		1	13.4	7.6	2.8	37	2.7	35

Table 20.19: Average variable query user interactions of all test series

- The main component of the JADE debugger is the diagnosis engine, which should eliminate as many incorrect diagnoses as possible. Some further tests in the context of the diagnosis process are discussed in Section 20.1.3.
- In particular, further tests should be carried out with longer and more sophisticated methods. These tests should include further source code structures and new fault classes.
- Tests should also be performed with full versions of both models, the *DFDM* and *ETFDM*. However, this can only be done, if the JADE modeling module can provide full versions of the *ETFDM*, which is currently not supported.
- In future versions of the JADE debugger different measurement selection algorithms should be implemented and tested in order to minimize the number of variable query interactions.
- Further tests should be carried out with the concurrent specification of multiple variable values in a single evaluation step. This includes new forms of variable query interactions as discussed in Section 19.3.
- More efficient techniques should be developed for handling the hierarchical debugging process. This includes reducing the overhead introduced through the switching from one hierarchy level to another and the use of flatter models, which allow for the debugging of sub-models together with their super-models in a single diagnosis step.

The following chapter discusses some problems with the approaches described in this work and shows in what respects the current version of the JADE debugger can be improved.

Chapter 21

Discussion

This chapter is dedicated to a discussion about the approaches presented in this work and the empirical results given in Chapter 20. In particular, we discuss the main strengths and weaknesses of our diagnosis and debugging techniques and present some enhancements, which look promising for future research projects. Like in the last chapter, we split this chapter into two sections. First, we discuss problems arising in the context of the diagnosis process and show how this phase of the debugging process can be improved. Second, we take a look at the whole debugging process. We discuss, which role a model-based debugging tool might play in the future as part of an integrated software development tool.

21.1 Diagnosis

The computation of diagnoses (see Chapter 16) is the basis for an efficient limitation of the search space of bug candidates and constitutes the main element in an interactive and iterative debugging process (see Chapter 17). The performance of the diagnosis process directly depends on the underlying model. Part II of this work describes in detail the creation and properties of various kinds of FDMs. Empirical diagnosis results obtained from the use of these models are given in Section 20.1. However, one of the main questions is, which classes of source code faults can be located by using the approaches described in this work. In the following we try to answer this question.

21.1.1 Fault classes handled by the JADE debugger

In Section 2.2 we argue that the focus of this work is on the localization of source code bugs, which (1) become observable as failures or output errors and (2) manifest themselves as logical faults in the analyzed source code. This explicitly excludes compile-time and run-time failures as well as faults violating the syntax or semantics of our target programming language Java. Further restrictions on the analyzed fault classes are presented in Section 5.2. Furthermore, in Section 2.2 the class of source code faults analyzed in this work is divided into functional and structural faults. Functional faults are all faults, which result in a certain variable

storing an incorrect value in at least one possible evaluation trace. In particular, these faults include the use of incorrect operators or the specification of incorrect literals, such as integer or boolean constants. Since these faults do not alter the structure of the program and, in case of a successful test-case, always become observable through an incorrect variable value, faults belonging to this class can generally be found with the JADE debugging environment. Whereas *DFDMs* are designed to potentially locate all functional faults of a given program, *ETFDMs* can only handle a functional fault, if a test-case is present, which exhibits a failure as a consequence of the functional fault. Note that in the case of functional faults the created system description (see Chapter 15) does not differ from the system description of the correct program.

Structural faults, on the other hand, are source code bugs, which alter the structure of the underlying program. This is the case, if the dependency graph [14] of the program is not structurally equivalent to the dependency graph of the correct program. The result of these faults is that the system description differs from the system description obtained by the correct program. From a global view structural faults result in an incorrect structure of the created system description. This means that either the ports of individual system components are incorrect or the connections linking multiple components are set in an incorrect way. Note that due to the hierarchical modeling used in this work, a structural fault can also lead to a system description with an incorrect structure at a lower level of the hierarchy. This can result in a system description at a higher model level, which is structurally correct, but includes a component with an incorrect behavior. In this case the incorrect system description structure is hidden behind the abstract behavior of the component, which includes the incorrect sub-system. When diagnosis is performed at statement level, structural faults occur in one of the following forms:

- Missing statements
- Superfluous statements
- Statements including structural faults, such as an access to an incorrect variable.

Missing statements are currently not handled by the JADE system. However, missing statements mostly occur in very special contexts, such as a missing termination condition in a recursive method or a missing incrementation/decrementation statement at the end of a loop body. These faults in most cases lead to the non-termination of the program or could be located by the use of special-purpose models suited specifically to these fault classes. Generally, a missing statement seems unlikely. Superfluous statements are detected by the JADE debugging environment, if they influence the value of an output variable of the debugged method. In these cases a superfluous statement is part of the computed diagnoses and can thus be located by the JADE system. Again, it seems unlikely that superfluous statements, which alter the value of variables, occur in a method's source code.

By far the most difficult problems arise in the context of statements, which are themselves structurally incorrect. In particular, these faults include an access to an incorrect variable. Since these faults result in a system description, which differs from the system description of the correct program, they cannot be located in the general case. The following program fragment demonstrates this behavior:

```
1.    int x=0;
2.    int y=0;
3.    x = 2;
4.    x = 2;    // should be y=2; expected results: x=2, y=2
```

Using any of the models presented in Part II of this work, the JADE debugger returns statement 2 as the only possible culprit of the error observed for variable y. However, the fault in statement line 4 cannot be located. In some cases, e.g., when line 4 reads `x=3;` and we expect the outputs x=2 and y=3, the debugger is able to locate the bug, but in general more powerful solutions need to be applied, such as the introduction of replacement fault modes for assignments [43].

Another problem arising in the context of structural faults are faults influencing the location structure of a given Java system. In particular, these faults appear in two different forms: (1) Two variables of reference type point at different locations, i.e., objects, but should in fact reference the same object. (2) Two variables of reference type point at the same location, i.e., object, but should reference different objects. The latter case represents a common bug, which is demonstrated in the following source code fragment:

```
1.    p1 = new Point(0,0);
2.    p2 = p1;              // should be p1.copy()
3.    p1.x = 1;             // expected result: p1=(1,0), p2=(0,0)
```

When using full FDMs to compute diagnoses, the debugger distinguishes between variables of reference type and locations. This is how the JADE debugger solves the aliasing problem (see Sections 7.5 and 8.3). In the above example, we specify $nok(p2)$ and get two bug candidates, i.e., statements 1 and 2. In case of the *SFDM* this distinction is no longer made, but object structures are used instead (see Chapter 14). The aliasing problem is solved implicitly by a FD for *p2* in statement 3 (see Section 14.4). In the above example all three statements are bug candidates. If we eliminate the FD for *p2* from statement 3, the dependency-based model delivers only statements 1 and 2 as possible culprits, but can no longer deal with aliasing. One solution is to use two different models, one with and one without aliasing. The best model to be used for a specific problem can then be chosen by the user (if a priori knowledge about the problem exists) or by the use of given error class statistics.

21.1.2 Enhancing diagnosis using FDMs

As already mentioned the underlying model directly influences the diagnosis performance of the JADE debugger. In this work FDMs are used during all diagnosis and debugging sessions. Section 20.1 presents empirical results from diagnosis

```
      void demo() {
        int tmp, a, b, c;
1.      tmp = 3;
2.      a = f(tmp);
3.      b = f(tmp);
4.      c = f(tmp);
      }
```

Figure 21.1: Example method *demo()*

sessions performed with different kinds of FDMs. This section discusses some enhancements, which could make the diagnosis process in combination with FDMs more efficient.

In certain cases the JADE environment cannot locate a given fault, when using the *ETFDM*. This can be explained by looking at the following source code fragment:

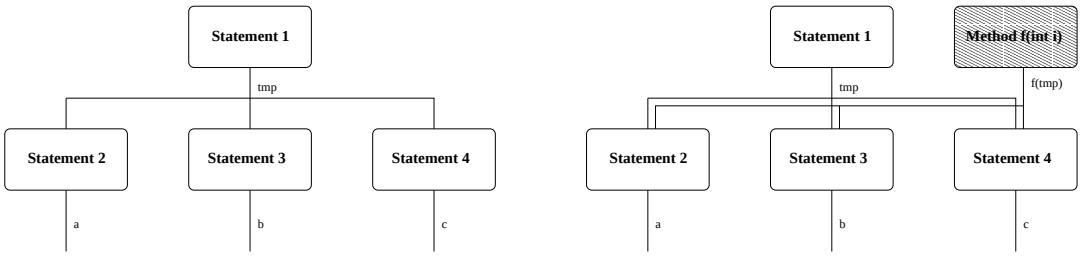
```
n.    while(i > 1) {
        // should be while(i > 0) {
n.1     x = a + b;
        ...
      }
```

Consider an evaluation trace, where at statement n variable i evaluates to one. Then the body of the **while** loop is not executed, although the condition should evaluate to `true` in a correct version of the method. If we compute the *ETFDM* for this source code, no FDs arise from statement n and this statement can thus never be part of a diagnosis. The solution to this problem is to explicitly incorporate the following fact into the model: *The values of all variables possibly modified in the loop body stay unaltered, if the condition is correct.* This can be achieved by adding FDs of the following form to the model (variable occurrences are stated without indices):

$$x \leftarrow \{x, c_1, ..., c_n\}$$

where x is a variable modified in the loop body and $\{c_1, ..., c_n\}$ is the set of all variable occurrences used in the condition of the loop. In the above example, we add the FD $x \leftarrow \{x, i\}$ to the *ETFDM*. The effect of this modification is that now the loop statement is a diagnosis, if variable x is observed to be incorrect at the method's output. The disadvantage of this approach is that in other cases, where the fault cannot be found in the loop condition, too many bug candidates are computed.

Another interesting behavior of the JADE debugger, which becomes apparent during experiments performed with the tool, is that it cannot distinguish between an incorrect method call and a buggy method being called. This is demonstrated by method *demo()*, which is depicted in Figure 21.1. The standard system description of method *demo()* (see Chapter 15) is depicted in Figure 21.2 (a). The



(a) Standard system description (b) Using meta-components

Figure 21.2: System description of method *demo()*

logical representation of *demo()* reads as follows (inconsistency sentences not stated):

$$\neg AB(s1) \rightarrow ok(tmp)$$
$$\neg AB(s2) \wedge ok(tmp) \rightarrow ok(a)$$
$$\neg AB(s3) \wedge ok(tmp) \rightarrow ok(b)$$
$$\neg AB(s4) \wedge ok(tmp) \rightarrow ok(c)$$

If we specify all output variables to be incorrect, i.e., $nok(a) \wedge nok(b) \wedge nok(c)$, we get the following conflict sets: $\{s1, s2\}$, $\{s1, s3\}$, and $\{s1, s4\}$. Clearly, this results in one single-diagnosis, i.e., $\{s1\}$, and the more unlikely diagnosis $\{s2, s3, s4\}$. The latter diagnosis says that statements 2 to 4 are all incorrect, no matter, whether they call the wrong method or call a method including a bug. However, this behavior is somewhat unexpected for the user, since a single bug in method *f(int i)* explains the whole misbehavior of method *demo()* and seems at least as likely as statement 1 being the culprit.

The solution to this problem is straightforward. We introduce a meta-component in the system description of *demo()*, which models the called method *f(int i)*. This component is independent of the components modeling statements 2 to 4, because the latter components only stand for the method call in contrast to the method declaration. The additional information needed to introduce these meta-components is already available in the FDMs created in Part II of this work, because these models include the signatures of the methods, on which a particular variable occurrence depends.

In the above example we create a new component, labeled *f(int i)*, and connect it to the components associated with statements 2 to 4 via a new connection labeled *f(tmp)*. The new system description is depicted in Figure 21.2 (b). The resulting logical model specifying the behavior of the new system description reads as follows:

$$\neg AB(s1) \rightarrow ok(tmp)$$
$$\neg AB(\text{f(int i)} \rightarrow ok(f(tmp))$$
$$\neg AB(s2) \wedge ok(tmp) \wedge ok(f(tmp)) \rightarrow ok(a)$$

```
     void vbm() {
       int tmp, a, b;
1.     tmp = 3;
2.     a = 2*tmp;
3.     b = 3*tmp;
     }
```

Figure 21.3: Example method *vbm()*

$$\neg AB(s3) \wedge ok(tmp) \wedge ok(f(tmp)) \rightarrow ok(b)$$

$$\neg AB(s4) \wedge ok(tmp) \wedge ok(f(tmp)) \rightarrow ok(c)$$

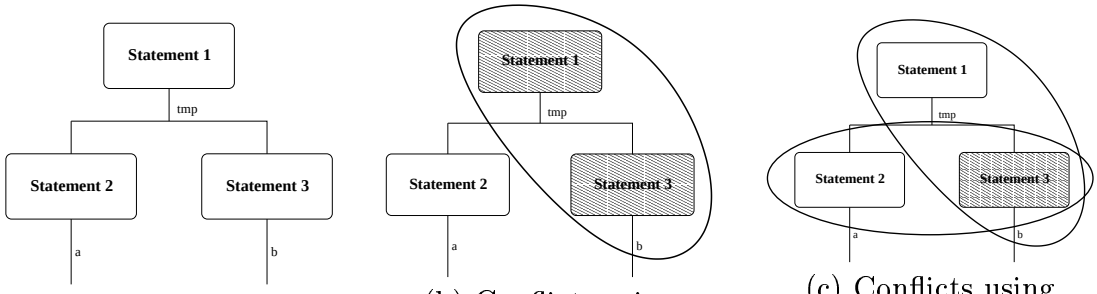
If we, again, perform a single diagnosis step, we now get the following conflict sets: $\{s1, s2, \text{f(int i)}\}$, $\{s1, s3, \text{f(int i)}\}$, and $\{s1, s4, \text{f(int i)}\}$. The resulting minimal diagnoses are $\{s1\}$, $\{s2, s3, s4\}$, and $\{\text{f(int i)}\}$. The last diagnosis solely explains the system's misbehavior as expected by the user. This is done by separately modeling method calls and method declarations.

21.1.3 Using alternative models

Another possibility to increase the diagnosis performance of the `JADE` debugger is the use of additional model types. These models can either be general models (like FDMs) or models, which focus on specific fault classes. The latter models are special-purpose models, which can be used by the user under certain circumstances, e.g., if a priori knowledge about the problem domain or the bug exists. In this section we concentrate on a general-purpose model type, which can be used instead of the FDMs presented herein: the Value-Based Model (VBM).

As already mentioned FDMs are designed in a way that they only allow for reasoning from a method's inputs to its outputs. Let us look at method *vbm()*, which is depicted in Figure 21.3. By using a FDM we might specify $ok(a) \wedge nok(b)$, which leads to the conflict $\{s1, s3\}$ and consequently to two single-diagnoses, i.e., $\{s1\}$ and $\{s3\}$. Figure 21.4 (a) shows the system description of method *vbm()* and Figure 21.4 (b) depicts the conflict produced by the use of a FDM.

Logically, statement 1 cannot be the culprit, if we use concrete variable values and know that statement 2 is correct. In other words, if statement 2 has the correct value of 6, then the diagnosis $\{s1\}$ alone does not explain the misbehavior of the system, because then statement 2 would have to be in an incorrect state, too. Therefore, from the user's point of view we only expect one single-diagnosis, i.e., $\{s3\}$. This can be achieved by VBMs, which make use of concrete values instead of *ok* and *nok*. Furthermore, VBMs allow for propagating these values from the inputs to the outputs and in at least some cases also from the outputs to the inputs of the system. VBMs thus are able to produce additional conflicts, which in turn lead to fewer diagnoses being computed. Figure 21.4 (c) shows the conflicts computed for the example method depicted in Figure 21.3. Note that the additional conflict is a *horizontal* conflict, which cannot be produced by



(a) System description (b) Conflicts using FDMs (c) Conflicts using VBMs

Figure 21.4: System description of method *vbm()*

FDMs, which only make use of *vertical* conflicts. As a consequence, $\{s3\}$ remains the only single-diagnosis.

Multiple VBMs have been incorporated into the `JADE` debugger and tested on various `Java` methods. [41] gives a brief description of one VBM and first results of experiments with this model type. The results show that in general the VBM is more powerful than FDMs and thus computes less diagnoses. Unfortunately, it is also computationally more complex and can so currently only be applied to small test applications. Future research will have to provide more efficient implementations of the VBM and further test its performance, especially in comparison with FDMs.

21.1.4 Outlook

This work shows, how various kinds of FDMs can be constructed and used to compute diagnoses for buggy `Java` methods. Section 20.1 gives some empirical results obtained from diagnosis sessions with the `JADE` debugging environment. There, the conclusions are that the pursued diagnosis approach efficiently reduces the search space by eliminating significant parts of the analyzed source code from the debugging scope by showing that these parts cannot account for an observed output error. Nevertheless, the obtained results indicate that the techniques described herein are not strong enough to exactly locate a bug location in a single diagnosis step.

This chapter includes some enhancements to the diagnosis process, which reach from small modifications of the used models to the use of alternative, more powerful models. As a matter of course, these enhancements are not the only ones, which can be designed and implemented for a more efficient diagnosis process. So far it has shown that a wide variety of different models can be used for the computation of diagnoses. All these models have individual strengths and weaknesses and are applicable to different problem domains and fault classes. For example, FDMs can be seen as general-purpose models, which can be constructed very quickly and seem to scale up for medium to large-size programs quite well. VBMs, on the other hand, are much slower to compute and can currently hardly be used in practical debugging sessions due to their slow response times. How-

ever, they are more powerful models than FDMs in the fact that they compute less diagnoses and more efficiently reduce the search space of bug candidates.

Therefore, future research has to concentrate not only on improving existing models and developing additional (special-purpose) models, but mainly on bringing these individual models together. This can, for example, be done by using different models in a single debugging session. The following sections discuss the current debugging performance of the JADE tool and show how this performance can be improved. Finally, we present some ideas of incorporating multiple models and debugging techniques into a single software development tool, which provides a maximum support for the user by an efficient combination of these debugging elements.

21.2 Debugging

The debugging process, which is depicted in Figure 17.1, is based on the computation of diagnoses and is designed to support the user in an optimal fashion until a certain fault is exactly located. Empirical debugging results obtained from experiments with the JADE debugger are given in Section 20.2. These results indicate that in most cases the JADE debugger significantly reduces the amount of user interaction needed in comparison with traditional debugging tools and thus provides a powerful tool for software developers. However, the debugging approaches presented herein also have a couple of drawbacks and existing problems. This section first presents some enhancements to the debugging process and shows, how the JADE debugger might be made more effective in future versions. Finally, we discuss which role model-based debugging techniques might play in the future and how these tools can be incorporated into existing software development tools.

21.2.1 Enhancing the JADE debugger

In Section 17.1 the main requirements of an interactive debugging tool are stated as follows: (1) User-friendliness, (2) response time, and (3) efficiency. These three criteria represent starting points for a systematic discussion about possible enhancements of the JADE debugger.

User-friendliness: The user-friendliness of the JADE debugger mainly depends on its GUI, which lets the user specify observations and displays the current sets of diagnoses and conflict sets. Although some effort has been made to construct an intuitive and clear GUI, this part of the JADE system is one, where future research could substantially improve the support for the user. The following ideas are thought as an outline for future research projects, which aim at increasing the user-friendliness of the JADE debugger:

- The first step in each debugging session is the specification of target output values. These values can either be specified interactively or by the use of the assertion language described in Section 18.1. The integration of the

JADE debugger into a more general software development tool can further increase the user-friendliness of the tool. In particular, information taken from the test phase can automatically be stored and used as input to the debugging process.

- The measurement selection algorithm currently computes the variable occurrence, whose evaluation eliminates a maximum amount of incorrect diagnoses. However, it does not take questions into consideration, which arise in the context of the system's user-friendliness. Examples of these questions are: *What knowledge about the underlying source code does a programmer have?* or *At which points within a program is it possible to make statements about the correctness of individual variable occurrences?* A more intuitive measurement selection can, for instance, rank variable occurrences higher, which are easier to specify (e.g., variables after loop statements), than variables, whose values are unlikely to be known by the programmer (e.g., a temporary variable in the middle of a complex computation). An alternative is to let the user specify potential measurement selection points, e.g., by using labels inserted into the source code (similar to assertions or breakpoints).

- Another key point in each debugging session is the specification of variable values. Currently, only one variable value is queried from the user, but it has already been proposed to let the user specify multiple values in one diagnosis step. When full FDMs are used during debugging, the states of concrete variable values have to be evaluated by the user. This is not always easy, especially, if the queried variable is a field of a remote object. This task becomes even more difficult, when *SFDMs* are used, which require the evaluation of whole object structures by the user in a single step. Improved visualization techniques could be used, which display (part of) the current object space and the internal state of individual objects in order to support the user during these tasks.

- Finally, the hierarchical debugging process can be made more intuitive by providing additional support, when a fault is located in a loop or selection statement. This is especially true for method calls, which currently can only be debugged, if the user manually selects the incorrect method call at expression level. Allowing for automatic debugging at expression level can make life easier for the user.

To sum up, there remain various ways of enhancing the JADE debugging environment as far as its user-friendliness is concerned. Most improvements include the creation of powerful GUIs, which provide the user with additional information about the state of the debugging process and the underlying source code. Furthermore, the support of psychologists could clarify, how programmers proceed in a fault localization session, what kinds of mental models they use, what knowledge about faults and the source code they have, and what their expectations about fault locations are. Together with the embedding of the debugger into more general software development tools, which is discussed in Section 21.2.2,

this could provide a substantial enhancement to the current version of the JADE debugger.

Response time: The FDMs described in Part II of this work can be constructed very quickly. This is especially true for non-recursive *DFDMs*, which model each method exactly once. In the recursive case, the creation of a *DFDM* with the fix-point algorithm presented in Chapter 13 is somehow more sophisticated and thus computationally more demanding. Nevertheless, experiments have shown that models for even complex recursive Java systems can be created with a simple prototype implementation in the range of a few minutes. Since the creation of an *ETFDM* requires an evaluation trace, the time needed to create an *ETFDM* directly depends on the time needed to execute the modeled method. Furthermore, the number of method calls and loop iterations of individual loops in the modeled method directly affects the modeling time of *ETFDMs*. Overall, all FDMs can be computed very efficiently, especially in contrast to value-based models. Once a model exists the debugging process can be performed without further model computations. Since the JADE debugger makes use of efficient diagnosis algorithms and a simplified measurement selection algorithm, response time in all experiments carried out lies in the range of a few seconds. By a more efficient implementation, response time could further be reduced and should not be the bottle neck of an efficient debugging tool.

Efficiency: The third criterion of a debugging tools is its efficiency. In Section 20.2 the performance of the debugger is evaluated as the number of user interactions during a debugging session. The following list presents some ideas of how to reduce this number of interactions and thus increase the debugger's efficiency:

- As already discussed in Section 21.1, the key element of the debugging process is the computation of diagnoses. There, various options of increasing the diagnosis performance are discussed, which should directly lead to a reduced number of variable queries in the debugging process.

- As proposed in Section 18.2, multiple test-cases can be used during a single diagnosis process to compute additional conflicts and thus less diagnoses. So far the use of multiple test-cases with the JADE debugger is only possibly in an off-line mode. By integrating this technique into the interactive debugging process the overall debugging performance of the JADE tool could be improved. This requires the adaptation of the existing measurement selection and variable query algorithms to the general case of multiple test-cases.

- The measurement selection algorithm used by the JADE debugger (see Section 17.2) is a quite simple algorithm, which should be improved in future versions of the JADE environment not only as far as its user-friendliness is concerned, but also with respect to its efficiency. One possibility to enhance the performance of the measurement selection algorithm is to use

fault probabilities of individual statements and expressions or to use additional run-time information of the analyzed program, such as absolute run-times of loop statements, etc... More sophisticated measurement selection algorithms could also take into account statistics about individual diagnosis components, which count the number of test-cases (in relation to all performed test-cases), in which the component appears as bug candidate.

- Another weak point in the debugging process is the high amount of interaction overhead introduced through the hierarchical modeling and debugging approaches pursued in this work. In order to locate faults nested in selection statements, loops, and called methods, special interactions have to be performed regardless of the used diagnosis and measurement selection algorithms. The results given in Section 20.2 indicate that in certain cases these interactions represent a significant percentage of the total amount of user interactions needed to locate a given fault. One alternative to reduce the amount of hierarchical interactions is to build flatter models. Section 10.3 briefly discussed the creation of flat models of selection statements. Similar, though more complex techniques can be used to create flat models of loop statements. Another possibility to use flat models is to completely unfold the target program. This includes the unfolding of all loop statements and the elimination of method calls by copying the body of the called method into the calling method. Clearly, this approach is only possible, if evaluation traces are used. The resulting programs can then be modeled and debugged with exactly the same techniques as described in this work. All diagnoses finally have to be mapped back to source code positions of the original program. Note that this approach is no longer trivial in cases, where the fault is duplicated as well (e.g., in the case of a fault in a loop). In these cases multiple-diagnoses have to be computed and mapped back to smaller (possibly single-) diagnoses.

In Section 21.1 the use of alternative models or special-purpose models, which are suited to specific fault classes, are proposed to enhance the diagnosis performance of the `JADE` debugger. Similar techniques can also be applied to the whole debugging process, which can be made more efficient through the concurrent or sequential use of multiple models. The following section shows, how multiple models can be used during a single debugging session.

21.2.2 An integrated software development tool

Finally, we discuss how the approaches presented in this work can be used in practical real-world applications. The answer to this question is to view the techniques presented herein from a more general perspective and combine them with (1) other techniques from the same field of research and (2) existing standard software engineering tools, which are currently used by programmers during the whole software life cycle.

The first goal is to create an integrated debugger in the way that multiple models can be used during a single debugging session. The advantages of this approach are twofold:

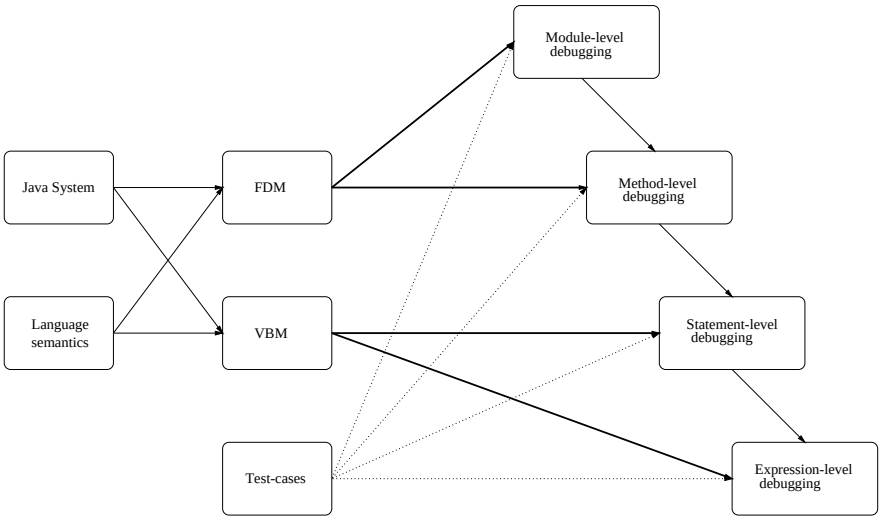


Figure 21.5: Using different models during the debugging process

1. Multiple models can be used during the debugging process one after another. For instance, FDMs are fast and relatively easy to create. They can thus be applied to even large applications. Nevertheless, their diagnosis performance is limited as demonstrated in Section 20.1. VBMs, on the other hand, are much more detailed and in general compute less diagnoses than FDMs. Unfortunately, they are computationally much more challenging and currently can only be applied to small programs. In an integrated debugging system, FDMs can be used to reduce the search space to an amount of remaining bug candidates, which then can be handled by the more detailed VBMs. It should be noted that this approach is not limited to FDMs and VBMs, but might as well include new model types. Figure 21.5 shows the architecture of an integrated debugging tool, which locates source code faults at module and method level by using FDMs. Once a particular method is found to locate the bug, VBMs are used for further debugging steps.

2. Multiple models can also be used in parallel. This means that depending on different fault classes specific special-purpose models can be created, which are then used during a fault localization process. The use of special-purpose models can be triggered by the user once he has certain expectations about the class or location of the bug. These special-purpose models can also be used in parallel to provide a wide range of alternative fault explanations to the user.

The second goal of an integrated debugging environment is to provide clear interfaces to existing software engineering and CASE tools. Only this guarantees a uniform and intuitive system, which is easy to use for software developers and widely accepted even by experienced programmers. We conclude this section by

presenting some synergies between a model-based debugging tool and standard software engineering tools:

- All initial variable observations needed for the automatic debugging process can directly be taken from the test phase. An integrated GUI has to be designed, which lets the user specify and run test-cases. In case of an observed output error or program failure a debugging session should automatically be initialized and started using information from the test phase.
- Existing debugging techniques, e.g., breakpoints and single-step operation, can be combined with model-based debugging approaches. One example for this combination is the use of assertions in the `JADE` debugger, which is presented in Section 18.1.
- Finally, future research may include automated software repair techniques. This means that once a fault is located by the debugging environment replacements for the faulty statement or expression can automatically be proposed to the user or tested for consistency by the system.

Chapter 22

Conclusion

In this work we have shown how model-based diagnosis techniques can be used to compute possible fault locations of buggy `Java` programs. Since this approach relies on the existence of a logical representation of the underlying `Java` system, large parts of this work deal with the creation of models of `Java` programs, which can automatically be created from the program's source code and easily be converted into a logical system description. We have presented three different model types, which are based on the collection of functional dependencies arising from the source code of the `Java` program. These functional dependency models differ in the amount of information used during their creation and the level of abstraction of the individual model components. In particular, the following model types have been introduced:

- The Evaluation Trace Functional Dependency Model (*ETFDM*) covers all functional dependencies, i.e., data and control dependencies, which arise during a particular program run, i.e., for a particular test-case. We have shown that by using an evaluation trace of the modeled method, all source code structures can be modeled non-ambiguously. This, in particular, holds for (polymorphic) method calls, loop and selection statements, and even recursive `Java` systems. We have argued that *ETFDMs* are sound models, which are minimal and complete in relation to a single program run. However, in most cases it is impossible to combine multiple *ETFDMs* to a single model covering all possible run-time scenarios, because of the immense amount of possible evaluation traces of a single `Java` method.

- The Detailed Functional Dependency Model (*DFDM*) computes all functional dependencies, which possibly arise at run-time during any possible program run. Since this model type is a purely static approach and does not make use of any run-time information, whatsoever, not all source code structures can be modeled without introducing a higher level of abstraction for various model components. In particular, we have presented algorithms for the static modeling of method calls, loops, and selection statements. Moreover, a fix-point algorithm has been proposed, which allows for the computation of static *DFDMs* of recursive `Java` systems. We have shown that in the general case the *DFDM* is no longer sound and can therefore be

seen as an approximation of a complete model, which covers all functional dependencies as they arise in any possible program run. We have shown that *DFDMs* are complete, but not minimal.

- The Simplified Functional Dependency Model (*SFDM*) is based on either the *ETFDM* or the *DFDM* and can be interpreted as a more abstract view of the underlying model. The *SFDM* is created by combining multiple variables and locations to object structures, which represent an abstract view of part of the analyzed `Java` system. Whereas the *SFDM* is easier to read and understand due to its simpler structure, it is less exact and detailed than its underlying model.

We have discussed in detail how individual source code structures can automatically be transformed into model fragments. Among others, we have described the modeling of variable assignments, method calls, selection statements, loops, arrays, strings, and whole `Java` methods. We have dealt with the main properties of all three model types in the context of the underlying source code structures and have shown how the different models handle polymorphic method calls, aliasing, and recursion.

Furthermore, in this work we have explained how functional dependency models together with standard model-based diagnosis techniques can be applied to the debugging of `Java` programs. In a first step we have shown how bug candidates can be computed. We have extended this approach to the creation of an interactive and iterative debugging tool, which combines the diagnosis process with efficient measurement selection and variable query algorithms and thus guides the user through a debugging session with a minimum of user interactions until eventually a single bug location is identified. We have introduced the `JADE` debugging environment, which is a prototype debugger implementing the basic concepts and techniques described in this work. The various types of user interactions and the exact handling of the `JADE` debugger have briefly been described.

The `JADE` debugging environment has been tested on various (buggy) `Java` methods, which include different source code structures and fault classes. In a first step the diagnosis performance has been evaluated to show, which parts of a `Java` method can be eliminated from the debugging scope during a single diagnosis step. Empirical results have shown that approximately 40% of all statements can be proven not to account for a given system failure and thus be eliminated in a single diagnosis step. In other words, only 60% of the analyzed source code remain for further debugging actions. We have performed all tests separately for the three model types introduced in previous section. Results have indicated that the *ETFDM* is superior to the *DFDM*, especially in case of selection and loop statements. The full models have been shown to perform slightly better than the respective *SFDMs*, what has to be contrasted with the easier handling and higher level of abstraction of *SFDMs*.

In a second step, we have tested the debugging performance of the `JADE` tool by comparing the amount of user interaction needed in comparison to the number of user interactions needed with a traditional debugging tool. Empirical results

have shown that significantly less user interactions are needed with the JADE tool. In particular, if *ETFDMs* are used as underlying models, on average only 37% of the user interactions of a traditional debugger are needed to exactly locate the installed source code fault. These results indicate that the debugging potential of the JADE debugging environment is very promising. Its performance should be increased in the future by performing additional tests and making use of improved measurement selection and variable query techniques.

Furthermore, we have discussed various strengths and weaknesses of the approaches presented in this work. We have discussed, which fault classes the JADE tool currently handles, and which fault classes still pose problems during the debugging process. Multiple possibilities of solving these problems and enhancing both, the debugger's diagnosis and debugging performance, have been discussed in detail. Finally, we have highlighted the future role of a model-based debugging tool like the JADE debugging environment in the more general context of the whole software development process. We have argued that the approaches presented herein should be combined with the use of alternative models (e.g., value-based models) to increase the performance of the JADE system and be embedded into a general software development tool in order to increase the user-friendliness and efficiency of such a tool and support the user during the whole software development process in an optimal way.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[3] Anton Beschta, Oskar Dressler, Hartmut Freitag, Michael Montag, and Peter Struss. A model-based approach to fault localization in power transmission networks. *Intelligent Systems Engineering*, 1992.

[4] G. W. Bond and B. Pagurek. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada, 1994.

[5] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis.* PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

[6] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, 38(3):31 – 41, 1995.

[7] Kai-Yuan Cai. *Software Defect And Operational Profile Modeling.* Kluwer Academic Publishers, 1998.

[8] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.

[9] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[10] Randall Davis and Walter Hamscher. Model-based reasoning: Troubleshooting. In Howard E. Shrobe, editor, *Exploring Artificial Intelligence*, chapter 8, pages 297–346. Morgan Kaufmann, 1988.

[11] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[12] D.Kranzlmüller, Ch. Schaubschläger, and J. Volkert. A brief overview of the mad debugging activities. In Mireille Ducasse, editor, *Proceedings of the Fourth International Workshop on Automated Debugging*, pages 229–234, 2000.

[13] Mireille Ducassé. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, pages 1–15, May 1993.

[14] 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*, 9(3):319–349, 1987.

[15] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[16] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[17] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[18] IEEE. IEEE Standard Classification for Software Anomalies (IEEE Std 1044-1993). Technical report, IEEE, 1989.

[19] IEEE. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Technical report, IEEE, 1990.

[20] Jerry R. Jackson and Alan L. McClellan. *Java by example.* Java Series. SunSoft Press, 1996.

[21] Bogdan Korel. PELAS–Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.

[22] Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.

[23] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1997.

[24] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. AI Support for Debugging Java Programs. In *3rd Workshop on Intelligent SW Eng.*, Limerick, Ireland, June 2000.

[25] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. JADE - A Step towards an Intelligent Debugger. In *Proc. DX'00 Workshop*, Morelia, Mexico, June 2000.

[26] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. JADE - AI Support for Debugging Java Programs. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligence*, Canada, November 2000. Also appears in [24].

[27] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Model-Based Debugging of Java Programs. In *Proceedings of the 4th International Workshop on Automated and Algorithmic Debugging, AADEBUG '00*, Munich, Germany, 2000.

[28] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java programs using a model-based approach. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, Scotland, 1999.

[29] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Locating bugs in Java programs – first results of the Java Diagnosis Experiments (Jade) project. In *Proceedings IEA/AIE*, New Orleans, 2000. Springer-Verlag.

[30] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling Java Programs for Diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, August 2000.

[31] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. A Value-Based Diagnosis Model for Java Programs. In *Proceedings of the Eleventh International Workshop on Principles of Diagnosis*, Morelia, Mexico, June 2000.

[32] Wolfgang Mayer. Modellbasierte Diagnose von Java-Programmen, Entwurf und Implementierung eines wertbasierten Modells. Master's thesis, Institut für Informationssysteme, Abteilung für Datenbanken und Artificial Intelligence, TU Wien, 2000. (only available in German).

[33] Glenford J. Myers. *The Art of Software Testing.* Wiley-Intersciene, 1979.

[34] Horst A. Neumann. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language (UML).* Carl Hanser Verlag, 1998.

[35] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[36] Inc. Object Management Group. OMG Unified Modeling Language Specification. Technical report, Object Management Group, Inc., 1999.

[37] Thomas Pawlin. Implementierung eines C-Übersetzers und Anwendung der modellbasierten Diagnose zum Software-Debugging von C-Programmen. Master's thesis, Institut für Informationssysteme, Abteilung für Datenbanken und Artificial Intelligence, TU Wien, 1996. (only available in German).

[38] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[39] Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts, 1983.

[40] Markus Stumptner, Dominik Wieland, and Franz Wotawa. Analysing Models for Software Debugging. In *Proceedings of the Twelfth International Workshop on Principles of Diagnosis*, Sansicario, Italy, 2001.

[41] Markus Stumptner, Dominik Wieland, and Franz Wotawa. Comparing Two Models for Software Debugging. In *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI)*, Vienna, Austria, 2001.

[42] Markus Stumptner and Franz Wotawa. Model-based debugging of functional programs. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, Cape Cod, May 1998.

[43] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, pages 1074–1079, Stockholm, Sweden, August 1999.

[44] Markus Stumptner and Franz Wotawa. Jade – java diagnosis experiments – status and outlook. In *IJCAI '99 Workshop on Qualitative and Model Based Reasoning for Complex Systems and their Control*, Stockholm, Sweden, 1999.

[45] R. Tarjan. Depth-first search and linear graph algorithm. *SIAM Journal of Computing*, 1(2), June 1972.

[46] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[47] Peter van der Linden. *Just Java.* Java Series. SunSoft Press, 1996.

[48] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[49] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[50] Franz Wotawa. *Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages.* PhD thesis, Technische Universität Wien, 1996.

[51] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation.* Academic Press, 2000.

Curriculum Vitae

Persönliche Angaben:

Name:	DI Dominik Wieland
Adresse:	Pötzleinsdorferstr. 80
	A-1180 Wien
	Austria
Tel. Büro:	+43-1-58801-18435
Fax.:	+43-1-58801-18492
Email:	wieland@dbai.tuwien.ac.at
Familienstand:	ledig
Staatsangehörigkeit:	Österreich
Geburtsort:	Salzburg
Eltern:	Mag. Gertraud Wieland, Mag. Elmar Wieland

Ausbildung:

- 1984 - 1992: Neusprachliches Gymnasium mit AHS-Maturaabschluß
- 1992 - 1999: Studium an der Technischen Universität Wien, Studienrichtung Informatik (Sponsion zum DI im März 1999)
- 08/1997 - 02/1998: Auslandssemester an der Middlesex University (London)
- seit 1993: Studium an der Wirtschaftsuniversität Wien, Studienrichtung Handelswissenschaften
- seit September 1999: Wissenschaftlicher Assistent und Doktorand am Institut für Informationssysteme, Abteilung Datenbanken und Artificial Intelligence

Tätigkeiten:

- 1992 - 1998: insgesamt 6 Monate Ferialpraxis bei Siemens PSE
- seit 09/1999: Wissenschaftlicher Assistent am Institut für Informationssysteme, Abteilung Datenbanken und AI der TU Wien