Recovering Software Tuning Parameters

by

Nevon Brake

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
June 2008

Copyright © Nevon Brake, 2008
Abstract

Autonomic Computing is an approach to designing systems that are capable of self-management. Fundamental to the autonomic ideal is a software’s awareness of and ability to tune parameters that affect metrics like performance and security. Traditionally, these parameters are tuned by human experts with extensive knowledge of parameter names and effects—existing software was not designed to be self-tuning. Efforts to automate the isolation and tuning of parameters have yielded encouraging results. However, the parameters are identified manually. This thesis proposes the adaptation of reverse engineering techniques for automating the recovery of software tuning parameters. Tuning parameters from several industrially relevant applications are studied for patterns of use. These patterns are used to classify the parameters into a taxonomy, and to develop a metamodel of the source code elements and relationships needed to express them. An extractor is then built to obtain instances of the relationships from source code. The relationships are represented as graphs, which are manipulated and queried for instances of tuning parameter patterns. The recovery is implemented as a tool for finding tuning parameters in applications. Experimental results show that the approach is effective at recovering documented tuning parameters, as well as other undocumented ones. The results also indicate that the tuning parameter patterns are not specific to a particular application, or application domain.
Acknowledgements

I would like to thank my supervisor, Dr. Jim Cordy, for his guidance and patience. His dedication to his students is unparalleled. I am grateful for the knowledge he shared, and the time, both personal and professional, he devoted to ensuring my progress and success.

I would also like to thank the present, and past, project participants from IBM: Liz Dancy, Marin Litoiu, Valentina Popescu, and Andrew Eberbach. Our discussions were always lively, insightful, and productive. A special thanks to Liz for overseeing things at IBM, for being an advocate, and for sharing her time and invaluable advice.

Lastly, I want to thank my family. Their support and encouragement lifted my spirit, and helped me to stay focused. To Alicia, I could not have done this without your infinite love and understanding.

This work was supported by the IBM Toronto Center for Advanced Studies (CAS), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Ontario Graduate Scholarship program (OGS).
# Contents

Abstract i  

Acknowledgements ii  

Contents iii  

List of Tables vi  

List of Figures vii  

List of Listings ix  

1 Introduction 1  

1.1 Motivation .................................................. 1  
1.2 Problem ..................................................... 2  
1.3 Terminology .................................................. 3  
1.4 Objective .................................................... 4  
1.5 Contributions ............................................... 5  
1.6 Organization ............................................... 6  

2 Background 8  

2.1 Autonomic Computing ........................................ 8  
2.2 Software Tuning Panels for Autonomic  
  Control ......................................................... 10  
2.3 Related Work ................................................ 13  
  2.3.1 Reverse Engineering ..................................... 14  
  2.3.2 Forward Engineering ..................................... 20  
2.4 Technologies .................................................. 23  
  2.4.1 Eclipse .................................................. 23  
  2.4.2 Relational Manipulation ................................ 26  
2.5 Summary ..................................................... 27
3 Overview
3.1 Empirical Study .................................................. 30
3.2 Fact Extraction .................................................... 30
3.3 Fact Manipulation .................................................. 32
3.4 Pattern Matching ................................................. 32
3.5 Summary ........................................................... 34

4 Empirical Study .................................................. 35
4.1 Applications ....................................................... 38
  4.1.1 Apache Tomcat .............................................. 38
  4.1.2 Mort Bay Jetty ............................................... 40
  4.1.3 Apache Derby ............................................... 42
  4.1.4 Oracle Berkeley DB Java Edition ......................... 43
4.2 Patterns .......................................................... 46
  4.2.1 Behavioural .................................................. 46
  4.2.2 Numerical ................................................... 51
  4.2.3 Statistical ................................................... 56
  4.2.4 Temporal .................................................... 62
4.3 Metamodel ......................................................... 67
  4.3.1 Classes ....................................................... 69
  4.3.2 Associations ................................................ 70
4.4 Summary ........................................................ 70

5 Fact Extraction .................................................. 72
5.1 Graph Schema .................................................... 73
  5.1.1 Nodes ........................................................ 74
  5.1.2 Edges ........................................................ 76
5.2 Source Code ....................................................... 77
  5.2.1 Parser ........................................................ 78
  5.2.2 Visitor ....................................................... 81
  5.2.3 Declarations ............................................... 83
  5.2.4 Flows ....................................................... 88
  5.2.5 Comparisons ............................................... 94
5.3 Summary ........................................................ 96

6 Fact Manipulation .................................................. 97
6.1 Method Invocations ............................................ 98
6.2 Polymorphism .................................................... 103
6.3 Assignments ..................................................... 108
6.4 Comparisons .................................................... 111
6.5 Conditions ....................................................... 113
# List of Tables

2.1 Semantic roles of variables ........................................ 15

4.1 Overview of the applications studied. ............................. 36

8.1 Overview of the additional applications. ......................... 129
8.2 Total number of facts extracted from each application. .......... 132
8.3 Results of the automated recovery for the applications. ......... 134
8.4 Percentage of undocumented parameters recovered that are considered spurious. ........................................ 135
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Overview of the STAC pipeline</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Structure of date analysis</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Components of the JMX architecture</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Class diagram of the core components in JDT</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Automating the recovery of software tuning parameters</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Extracting relationships from source code</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Manipulating the graph to create new relationships</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Pattern matching to identify tuning parameters</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Classification of tuning parameter patterns</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>Stereotype of the toggle classification</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Stereotype of the enumerable classification</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Stereotype of the accumulator classification</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Stereotype of the capacity classification</td>
<td>53</td>
</tr>
<tr>
<td>4.6</td>
<td>Stereotype of the counter classification</td>
<td>54</td>
</tr>
<tr>
<td>4.7</td>
<td>Stereotype of the threshold classification</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>Stereotype of the average classification</td>
<td>58</td>
</tr>
<tr>
<td>4.9</td>
<td>Stereotype of the maximum classification</td>
<td>60</td>
</tr>
<tr>
<td>4.10</td>
<td>Stereotype of the minimum classification</td>
<td>61</td>
</tr>
<tr>
<td>4.11</td>
<td>Stereotype of the delay classification</td>
<td>63</td>
</tr>
<tr>
<td>4.12</td>
<td>Stereotype of the duration classification</td>
<td>64</td>
</tr>
<tr>
<td>4.13</td>
<td>Stereotype of the period classification</td>
<td>65</td>
</tr>
<tr>
<td>4.14</td>
<td>Stereotype of the point classification</td>
<td>66</td>
</tr>
<tr>
<td>4.15</td>
<td>Stereotype of the timeout classification</td>
<td>67</td>
</tr>
<tr>
<td>4.16</td>
<td>Class diagram of the metamodel</td>
<td>68</td>
</tr>
<tr>
<td>5.1</td>
<td>Graph schema of the metamodel</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Class diagram of the parser and abstract syntax tree nodes</td>
<td>80</td>
</tr>
<tr>
<td>5.3</td>
<td>Class diagram of abstract syntax tree nodes and visitors</td>
<td>81</td>
</tr>
<tr>
<td>6.1</td>
<td>Relating variables of formal parameters to actual parameters</td>
<td>99</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.2</td>
<td>Relating indirectly returned values to methods.</td>
<td>100</td>
</tr>
<tr>
<td>6.3</td>
<td>Relating method return values to expressions in which the method is</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>an operand.</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Relating method return values to expression for which the method is</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>a condition.</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Relating methods that indirectly override one another.</td>
<td>104</td>
</tr>
<tr>
<td>6.6</td>
<td>Relating return values of methods that override one another.</td>
<td>106</td>
</tr>
<tr>
<td>6.7</td>
<td>Relating parameters of methods that override one another.</td>
<td>107</td>
</tr>
<tr>
<td>6.8</td>
<td>Relating the operands of assignment expressions.</td>
<td>110</td>
</tr>
<tr>
<td>6.9</td>
<td>Canonicalizing unary expressions, making implicit assignments explicit.</td>
<td>111</td>
</tr>
<tr>
<td>6.10</td>
<td>Relating operands of infix expressions having comparison operators.</td>
<td>113</td>
</tr>
<tr>
<td>6.11</td>
<td>Relating the operands of logical infix expressions to their condition's</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>expressions.</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>A weak relationship graph containing a toggle.</td>
<td>118</td>
</tr>
<tr>
<td>7.2</td>
<td>A weak relationship graph containing a counter.</td>
<td>120</td>
</tr>
<tr>
<td>7.3</td>
<td>A weak relationship graph containing a maximum.</td>
<td>123</td>
</tr>
<tr>
<td>7.4</td>
<td>A weak relationship graph containing a delay.</td>
<td>125</td>
</tr>
</tbody>
</table>
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Source code with markup around the tuning parameter declaration.</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Source code rearchitected to use a control panel.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Control panel generated by the TXL transformation.</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of a standard managed bean.</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Description of a managed bean from Apache Tomcat.</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Description of a managed bean from Mort Bay Jetty.</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Description of a managed bean from Oracle Berkeley DB Java Edition.</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Building abstract syntax trees in JDT using a parser.</td>
<td>80</td>
</tr>
<tr>
<td>5.2</td>
<td>Stack implementation for the visitor’s temporary storage.</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>Visiting a type declaration to extract overriding methods.</td>
<td>85</td>
</tr>
<tr>
<td>5.4</td>
<td>Visiting a field declaration to extract fields.</td>
<td>87</td>
</tr>
<tr>
<td>5.5</td>
<td>Visiting a method declaration to extract formal parameter variables.</td>
<td>88</td>
</tr>
<tr>
<td>5.6</td>
<td>Visiting an assignment expression to extract an explicit assignment.</td>
<td>90</td>
</tr>
<tr>
<td>5.7</td>
<td>Visiting a postfix expression to extract an implicit assignment.</td>
<td>90</td>
</tr>
<tr>
<td>5.8</td>
<td>Visiting a method invocation to extract parameters.</td>
<td>91</td>
</tr>
<tr>
<td>5.9</td>
<td>Visiting a method return statement to extract a return value.</td>
<td>92</td>
</tr>
<tr>
<td>5.10</td>
<td>Visiting an array access expression to extract a subscript.</td>
<td>93</td>
</tr>
<tr>
<td>5.11</td>
<td>Visiting a conditional expression to extract a condition.</td>
<td>93</td>
</tr>
<tr>
<td>5.12</td>
<td>Visiting an infix expression to extract comparisons.</td>
<td>95</td>
</tr>
<tr>
<td>7.1</td>
<td>Querying the graph for instances of the toggle pattern.</td>
<td>117</td>
</tr>
<tr>
<td>7.2</td>
<td>Querying the graph for instances of the counter pattern.</td>
<td>120</td>
</tr>
<tr>
<td>7.3</td>
<td>Querying the graph for instances of the maximum pattern.</td>
<td>122</td>
</tr>
<tr>
<td>7.4</td>
<td>Querying the graph for instances of the delay pattern.</td>
<td>124</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter introduces the motivation for this work. It outlines the research problem, the objectives, and the contributions of the thesis.

1.1 Motivation

The increasing complexity of computing systems is overwhelming the capabilities of developers and administrators to integrate and manage them [24]. Autonomic computing is an initiative introduced by IBM in 2001 to combat this complexity by designing self-managing systems. That is, systems that are capable of self-configuration, self-optimization, self-healing and self-protection. Since its inception, more than 400 product features in 36 distinct IBM products have been designed with autonomic computing capabilities [17]. But what about the myriad existing systems not designed for autonomic operation? Such systems are both an asset and a liability for companies. They have been evolved and specialized to cope with evolving requirements, all the while becoming entrenched as corporate cornerstones. Meanwhile, their complexity
has increased, making them more difficult to manage and maintain [21]. Manual efforts to rearchitect existing systems for autonomic control would encounter the same race against complexity already being faced by developers and administrators.

Software Tuning Panels for Autonomic Control (STAC) [9] is a project aimed at automatically rearchitecting software systems for autonomic control. An important aspect of autonomic computing is an application’s awareness of and ability to tune its parameters. Parameter tuning can be used to influence metrics like performance and security. Buffer pool sizes, for example, can be tuned to improve transaction throughput as a database application’s workload fluctuates [23]. Tuning of this sort is traditionally performed by human experts with extensive knowledge of parameter names and effects. For autonomic controllers to perform a similar function, the parameters must be exposed in some way. This is problematic as they are often scattered, and sometimes hidden. STAC works to overcome these issues by isolating the tuning parameters into a single architectural location. The result is a semantically equivalent system that provides localized access to its tuning parameters.

### 1.2 Problem

Previous work to develop a STAC prototype [9] for isolating tuning parameters in Java applications has shown promising results. Unfortunately, the prototype requires manual placement of markup around the variable declarations of tuning parameters. This markup indicates which variables STAC should isolate. Searching for the variables and adding the markup, even for a small system, can be tedious and error-prone. An automated identification mechanism is needed. In addition, it is has been shown [23]
that tuning of one parameter can impact others. It is thus desirable to identify relationships between parameters, or at a minimum, classify parameters into related groups. This thesis considers these issues within the context of the STAC prototype and its requirements.

1.3 Terminology

The terms *tunable* and *tuning* are used throughout this thesis. As used herein they have related but distinct meanings, though their usage in the related literature does not make such a distinction. The term *tunable* is used in [9] when referring to parameters that are meant to be tuned, as well as those that are not. For example, a parameter representing the capacity of a resource may be tuned, while a parameter representing the utilization of that resource may not. Both types are useful parameters for tuning, but referring to both as tunable can be confusing. Instead, only those parameters that are mutable are referred to as *tunable*. This is considered a special case of tuning. Parameters for tuning an application, mutable or not, are referred to as *tuning* parameters.

The term *parameter* is often used in computer science to mean a reference, or value, passed to a procedure or function. Here, it is used as a synonym for a variable or object property that affects or measures some aspect of the behaviour of an application. More specifically, it refers to scalar properties of objects. The term *object* is meant in the object-oriented sense, with the understanding that the target language is Java.
1.4 Objective

Some open questions that this thesis addresses are:

- How can existing reverse engineering techniques be used to recover tuning parameters?
- How can tuning parameters be classified to improve cohesion?
- How can implicit tuning parameters be made explicit?

The response to the first question involves surveying the existing techniques and tools. For nearly two decades researchers in the reverse engineering and software analysis communities have been developing techniques and tools to facilitate the maintenance and evolution of legacy systems. These efforts have focused on tasks such as fact extraction, dependency analysis, pattern recognition, and impact analysis. As such, the process usually begins by extracting information from software artifacts like source code, documentation, and revision history. The information is then analyzed for relationships, dependencies and patterns. Once the analysis is complete, the results are used to derive new, more abstract relationships. It is surmised that a similar process of extraction and analysis can be used to recover tuning parameters.

There are two aspects to the second question. Firstly, there is the aspect of a method for classifying—given a classification scheme, how can the parameters be divided into cohesive groupings? There is a presumption here that such a classification scheme already exists. This leads to the second aspect. Since a classification scheme does not exist, how can one be created and what information should it contain?

The last question is perhaps the most ambitious to address. Variables, in the syntactic sense, are explicit source code elements that can be readily extracted for
analysis. Unfortunately, not all tuning parameters are conveniently declared as variables. In an object-oriented application, some may be properties of classes for which the source code is inaccessible. An example of this is the size property of the collection types in Java. Resource pools often use collections to maintain object references. Therefore, the ability to adjust and monitor properties of these collections is desirable. However, this must be accomplished indirectly through method invocation. Direct variable access is neither feasible nor desirable due to encapsulation. Even more abstract are properties for which there are no variables or methods. Consider a tree data structure. The depth and breadth of a tree could be useful for tuning. While it is possible to compute them using functions, it would be more efficient to expose them as explicit tuning parameters.

1.5 Contributions

This thesis makes several contributions to the areas of reverse engineering and autonomic computing: a taxonomy, a methodology, a metamodel, and a prototype tool.

It is helpful to first characterize what makes a parameter suitable for tuning. To help with this characterization, a taxonomy of tuning parameters is developed from observing how they are manifested in the source code of existing applications. This empirical study focuses on server-oriented applications, particularly those from the database management and web serving domains. The tuning parameters are classified according to how they are referenced in the source code.

Using ideas and techniques from previous work on reverse engineering and impact analysis, a methodology is demonstrated for automating the recovery of software tuning parameters. Starting with a documented set of tuning parameters, the source code
manifestations of them are analyzed for patterns. Based on the observed characteristics of their use, a metamodel of source code elements and relationships is developed. The metamodel serves as the basis for the extraction of relationships from source code. These relationships are represented as graphs, which are manipulated and queried for instances of tuning parameter patterns.

Finally, to demonstrate the effectiveness of the methodology, a prototype tool is implemented and evaluated against several applications. The evaluation compares the results of the automated tool against the documentation. The tool’s objective is to recover those tuning parameters that the applications’ developers have documented.

1.6 Organization

The remaining chapters are organized according to the following topics:

- Chapter 2 discusses some related work, and provides background information that is important for understanding the subsequent chapters.

- Chapter 3 provides an overview of the approach to automating the recovery of software tuning parameters.

- Chapter 4 looks at how tuning parameters are manifested in source code. The empirical study examines the documented tuning parameters from several industrially relevant applications. Based on patterns of use, the parameters are classified into a taxonomy. The characteristics of the patterns are also captured in a metamodel that serves as the basis for the analysis.

- Chapter 5 describes how the metamodel is used for representing relationships
between source code elements as graphs, and how the graphs are extracted for analysis.

• Chapter 6 describes how the extracted graphs are manipulated to simplify the pattern matching process.

• Chapter 7 describes how the graphs are queried for instances of the tuning parameter patterns.

• Chapter 8 discusses the effectiveness of the approach in terms of its accuracy, and provides experimental evidence that the tuning parameter patterns are not specific to a particular application, or application domain.

• Chapter 9 summarizes the motivation for the thesis, states some conclusions, and hints at possible future work.
Chapter 2

Background

This chapter provides some background information, and discusses some related work in the area of reverse engineering for variable classification. It also introduces the existing technologies that are leveraged by this work.

2.1 Autonomic Computing

The concept of autonomic computing was introduced by IBM in 2001 to combat the complexity of managing and maintaining software systems. Kephart and Chess [20] describe the vision of autonomic computing as "...the grand challenge to create self-managing computing systems." As the complexity and ubiquity of computing systems continue to increase, administrators, and developers alike, are in a race to keep up. These systems are no longer single, solitary pieces of hardware and software. They are conglomerations—systems of systems—woven together with communication protocols and interconnected by diverse network infrastructures. This heterogeneity makes configuration and problem diagnosis difficult. Autonomic computing is meant to ease
the burden by making systems more self-aware, and hence self-governing. Underlying the autonomic vision are the four aspects of a self-managing system [20]:

- **Self-configuration**: Systems self-configure based on policies that specify high-level, goal-oriented objectives for what is desired but not how to accomplish it. New components register themselves and their capabilities. Based on this information, systems react accordingly by taking into account their composition and configuration.

- **Self-optimization**: Systems self-optimize by continually and proactively seeking to upgrade their function and make themselves more efficient in terms of performance and cost. This means applying patches and updates automatically, and monitoring and experimenting with their own parameters for tuning.

- **Self-healing**: Systems self-heal by detecting, diagnosing and repairing problems stemming from bugs or external failures. Data are gathered from log files and monitoring agents and used to match diagnoses and automatically apply patches. Regression suites can even be used to test the patched system to ensure proper functioning.

- **Self-protection**: Systems self-protect against attacks and cascading failures. Early warning sensors alert systems to possible problems allowing them to take preventative or mitigative action. Self-protection is related to self-healing since problems left uncorrected by the latter can result in cascading failures.

The approach for realizing the autonomic vision is fundamentally an architectural one. It is theorized that by composing *autonomic elements* together—supported by
a distributed, service-oriented infrastructure—system self-management will emerge. The term *autonomic element* is used to describe a set of managed elements like devices, software resources or even applications, that are combined with an autonomic manager that controls and represents them. An *autonomic element* is responsible for its own state and behaviour, and is driven or assisted by the other elements with which it interacts.

### 2.2 Software Tuning Panels for Autonomic Control

Dancy and Cordy [9] developed the Software Tuning Panels for Autonomic Control (STAC) prototype for automatically rearchitecting legacy systems to facilitate autonomic management. The approach has two main steps. Tunable parameters in the legacy system are first identified by manually placing markup around their declarations. STAC then automatically traces all references to these parameters and redirects them to a newly generated component called the *control panel*. The target language for the prototype is Java, and so the control panel is realized as a new class.

The process starts by merging the program’s source files into a single file, delimited by tags. These tags are later used to split the source code back into its original file and package structure. The merged source is analyzed and rearchitected using a series of source transformation rules written in the TXL [8] programming language. The rules perform the reference tracing, control panel generation and reference redirection. An overview of the STAC pipeline is shown in Figure 2.1 on the next page.

Parameters are injected into the control panel as inner classes bearing the same
CHAPTER 2. BACKGROUND

Figure 2.1: Overview of the STAC pipeline [9]

name. The inner class provides methods for initializing, setting, and retrieving the value of the parameter. Note though that a separate inner class must be created for each copy of a variable that is explicitly created in the original source. Otherwise, the same instance of the parameter, or value in the case of primitives, would be shared by all instances of the referring class. References to each parameter are transformed from the usual initializers, accessors and assignments to method invocations on the corresponding control panel inner class. The transformation rules ensure semantic equivalence and syntactic validity. Listing 2.1 on the next page shows a simple server class with markup around the `maxConnections` variable. Listing 2.2 on the next page shows the result of the STAC transformation. The `maxConnections` variable has been isolated to a control panel (Listing 2.3 on page 13) and the initialization and accessor references have been redirected to the appropriate control panel methods.
Listing 2.1 Source code with markup around the tuning parameter declaration.

```java
public class Server {
    <control_param>
    private int maxConnections = 100;
    </control_param>
    private int numConnections = 0;

    public boolean accept() {
        if (numConnections >= maxConnections) {
            /* Reject connection. */
            return false;
        }
        /* Accept connection. */
        ...
    }
    ...
}
```

Listing 2.2 Source code rearchitected to use a control panel.

```java
public class Server {
    CPanel.maxConnections.set(100);
    private int numConnections = 0;

    public boolean accept() {
        if (numConnections >= CPanel.maxConnections.get()) {
            /* Reject the connection. */
            return false;
        }
        /* Accept the connection. */
        ...
    }
    ...
}
```
Listing 2.3 Contral panel generated by the TXL transformation.

```java
public class CPanel {
    public static class maxConnections {
        public static int maxConnections;

        public static int create() {
            return maxConnections;
        }

        public static int set(int value) {
            maxConnections = value;
            return maxConnections;
        }

        public static int get() {
            return maxConnections;
        }
    }
}
```

2.3 Related Work

This section describes related work in the areas of reverse and forward engineering. Reverse engineering is the process of analyzing a system to: identify its components and their inter-relationships, and create different representations or abstractions [6]. Forward engineering refers to the more traditional software development methodology of moving from requirements to a design and from a design to an implementation. Reverse engineering itself does not actually change the system being analyzed. This distinguishes the proposed research for STAC from previous work.

Maintenance activities require developers to understand what a system does and how it does it. Trying to understand source code directly is arduous and time-consuming. The understanding task is facilitated by recovering information about data abstractions, control flows and data flows. Identification of tuning parameters
can be regarded as a maintenance activity requiring a similar kind of program comprehension. Here, the understanding task is related to the implicit or explicit intent of variables. Consider, for example, that the sizes of data structures are often limited by some threshold value. The value is stored in a variable and given a name like `maxSize`. It is syntactically akin to other variables of its type, but the informal knowledge represented in the name provides an important clue as to its intent. Likewise, usage of a variable in the condition of a branch or loop statement can indicate its intent as a threshold.

Formal methods for conducting static analysis have been researched extensively and have produced numerous techniques for data-flow and control-flow analysis. While useful, they are not specifically discussed here. The focus is instead on those analyses that are specific to variable roles and variable classification. These analyses are for the most part based on legacy C and COBOL systems, but elements of the ideas are nonetheless transferrable to object-oriented languages like Java.

### 2.3.1 Reverse Engineering

**Conceptual Roles of Data**

Deng and Kothari [11] describe a tool designed to help associate program variables with their conceptual roles and the inter-relationships among such roles. The analysis is divided into two levels: program-centric and knowledge-centric. The program-centric analysis extracts syntactic and semantic information about program variables using inter-procedural analysis. The knowledge-centric analysis provides a rule-based mechanism that must be customized for a given domain.

Variable roles are represented by a series of feature vectors, wherein each vector
### Table 2.1: Semantic roles of variables [11]

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Used in a file operation.</td>
<td>Loopcontrol</td>
<td>Used in a loop control head.</td>
</tr>
<tr>
<td>Input</td>
<td>Used in an input statement.</td>
<td>Loopbody</td>
<td>Used in a loop.</td>
</tr>
<tr>
<td>Output</td>
<td>Used in an output statement.</td>
<td>Subscript</td>
<td>Used as an array subscript.</td>
</tr>
<tr>
<td>Read</td>
<td>Read in an expression.</td>
<td>Write</td>
<td>Used to save an expression result.</td>
</tr>
<tr>
<td>Return</td>
<td>Used to carry a return value.</td>
<td>Ifcontrol</td>
<td>Used in a branch control.</td>
</tr>
<tr>
<td>Param</td>
<td>Used as a parameter.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

contains the semantics of an individual use of that variable. The semantic roles are given in Table 2.1. Every use of a variable is associated with an individual feature vector. The analysis is recursive, bringing all uses of a variable to the top-level. For example, if a variable is passed as a function parameter, the uses of the parameter are tracked and brought back as uses of the original variable. The feature vectors, once constructed, are used to perform data relation analysis to refine the semantic role information. Here, local data dependencies are extracted. If a feature vector contains the role *Loopcontrol* the corresponding upper-bound and lower-bound are extracted. These local relationships are used to infer other relationships. If a subscript to an array is bounded by some variable, then transitively the array itself is related to that bound.

At the next level, the concepts can be annotated to specialize the analysis for a particular domain based on a set of rules. These rules are developed from expert knowledge and executed as scripts. At this level, there are three categories of analysis: itemized, synthetic, and crosscutting. Itemized analysis is essentially localized
pattern recognition. It does not consider control flow. If a variable is used as a subscript in a scientific application then it might be annotated with the concept *space dimension*. Synthetic analysis is used to summarize roles of a variables or obtain a global view of variable uses. Dependencies are not considered. Crosscutting analysis considers control-depenence and data-dependence. The objective is to infer conceptual or semantic roles from other relevant variables. They use the example of scientific computing, wherein a branch statement is commonly used either for bounds checking or threshold comparison. The two uses can be disambiguated using a rule that determines whether all relevant variables in the same branch statement have been annotated *boundary* or not.

**Variable Classification**

Dean et al. [10] developed a variable classification technique to assist with the maintenance of legacy COBOL systems for the year 2000 problem. The objective was to identify variables representing dates, and the format of those dates. This information could then be used by developers to fix potential faults beforehand.

In their method, a set of base facts are extracted that contain the type and storage allocation of variables, and the relationships between variables based on assignments and comparisons. While other classification approaches [19] make use of inter-module relationships, they found that assignment and comparison facts were sufficient. This is attributed to their analysis being applied to the entire application at once. Every variable declaration included in multiple programs is clustered and treated as a single entity. Note however that their concept of assignment is abstracted to include statements besides *MOVE*. In their simplified model, input using *ACCEPT*, for example,
The analysis begins by creating an initial set of date variables based on naming conventions. This uses the declared name and the size of the allocated storage to identify variables that are most likely to represent dates. The naming convention rules are prioritized and can be applied as patterns or anti-patterns to various parts of variable identifiers. The initial variable set is then expanded by tracing references between variables. This uses the assignment and comparison facts gathered previously as opposed to the source code. The analysis is based solely on references so data flows are not considered. If the two variables are the same size, the reference is considered trivial. However, COBOL allows data transfer and comparison between variables of differing size. In this case, another level of inference must be applied to determine the types. This is particularly important for determining the format of dates. Assigning a variable containing a day, month and year to a variable for year reveals nothing about the format of the day and month. In the final analysis phase, structured variables are examined to infer their type based on the types of their individual members. The overall structure of the date analysis is shown in Figure 2.2.
Data-Centered Analysis

Joiner et al. [18] proposed focusing on variables and their inter-relationships instead of on their control structure. They argue business applications, especially, involve much more data movement and manipulation than sequencing, branching or looping. The approach starts by automatically reverse engineering a set of dependence graphs directly from source code to capture data and control dependence among variables. These graphs are then used for further analyses such as slicing and ripple effect analysis. They list the main features of their data-centered approach as:

- **Dependence analysis**: The dependence model is explicitly maintained to address data-related queries. The interest is on determining static dependence relationships between program entities. Analysis is at the variable level as opposed to the statement level since the goal is to determine which variables have an effect on or are affected by other variables.

- **Variable classification**: Variables are classified according to eight categories: domain, program, local, global, input, output, constant and control. Domain variables can be mapped to the application domain, while program variables cannot. Program variables are implementation specific, holding intermediate results, controlling execution flow, and so on. Local variables are confined to a single scope while global variables are not. Input variables are involved in reading from a file or device. Output variables are involved in writing to a file or device. Constants are variables whose value does not change. Control variables are used in branches or loops as predicates.

- **Slicing**: Slices are computed in four ways: program, variable, condition, and
domain variable. Program slicing finds statements that affect the values of a set of variables at a particular program location, called the criterion, to produce another program with the same behavior with respect to the criterion. Variable slicing is a variant of program slicing that finds variables, as opposed to statements, that affect the variables in the criterion. Condition slicing filters everything except predicates. Domain variable slicing filters non-domain variables from a variable slice. The slices are computed in either a forward or backward direction.

- **Ripple effect analysis**: The process of discovering and correcting the side-effects of changing a program. The slicing techniques described above are used to determine the statements or variables that are affected by a change. A change made with respect to one side-effect has other side-effects that become subject to the analysis. The analysis proceeds in an iterative manner until no further changes are needed.

Both data and control dependence between variables are captured in the dependence graphs. Data dependence is defined as one variable being used to define the other, while control dependence is defined as one variable controlling the value of the other. Using a parser, the graphs are constructed in two phases. First, the program is scanned to identify variables and their define/use information. A vertex is created for each variable. Data dependence edges are then created between two vertices whenever a use-define relationship exists between them. Likewise, control dependence edges are created between vertices that represent a predicate variable and all the vertices for variables within the block controlled by that predicate variable.
2.3.2 Forward Engineering

Java Management Extensions

The Java Management Extensions (JMX) specification defines an architecture, design patterns, API, and services for management and monitoring in the Java programming language [27]. It enables developers to instrument their code and implement both local and distributed management solutions. The architecture has three levels: instrumentation, agent, and distributed services. The relationship between components at the different levels is shown in Figure 2.3 on the next page. The instrumentation level specifies how resources are made manageable. A resource could be an application, a service, a device, and so on. The agent level specifies how resources are made available to management applications. The distributed services level specifies how agents coordinate and how management views are mapped to other protocols like SNMP or HTTP.

An application is made manageable by embedding a management object server and exposing the desired functionality as one or more managed beans. A bean is essentially a component that supports introspection of its properties. A managed bean has an interface that exposes its named and typed properties, named and typed operations, as well as typed notifications that can be emitted. To ensure all managed beans provide this instrumentation in a standardized way they must follow the design patterns and interfaces defined in the JMX specification.

By definition a managed bean is a concrete Java class that implements an interface of the same name but with the suffix MBean. Alternatively, it could implement the DynamicMBean interface. The former is referred to as a standard bean and the latter is a dynamic bean. It may also, optionally, implement the NotificationBroadcaster
interface. Properties of a standard bean are introspected from the interface getters and setters and operations are all the other methods. With dynamic beans, attributes and operations are exposed indirectly. Metadata for the properties and operations must be acquired separately and passed to a parameterized getter, setter or invoker. An example of a standard MBean implementation is shown in Listing 2.4 on the next page. The names of properties and operations are introspected according to a defined lexical convention for method names. The bean has two properties and one operation. The size property is read-only since there is only a getter, while capacity is read-write. The dump operation can be invoked by an agent through the bean’s registered object server. Beans are registered with the server using a unique name.

Aspects of managed beans are comparable to the control panel in STAC. Both
public interface ThreadPoolMBean {
    public int getSize();
    public void setCapacity(int capacity);
    public int getCapacity();
    public void dump();
}

public class ThreadPool implements ThreadPoolMBean {
    private Vector pool = new Vector();

    public int getSize() {
        return pool.size();
    }

    public void setCapacity(int capacity) {
        if (pool.capacity() <= capacity) {
            pool.ensureCapacity(capacity);
        } else {
            pool.setSize(capacity);
            pool.trimToSize();
        }
    }

    public int getCapacity() {
        return pool.capacity();
    }

    public void dump() {
        /* Dump the pool contents to standard output. */
        ...
    }

    /* Remaining thread pool methods... */
    ...
}
expose information about resources in an application. However, there are two significant differences. Managed beans are general purpose management components that support not only properties, but also operations and notifications. Additionally, while the control panel enforces architectural isolation of properties, the managed beans do not.

2.4 Technologies

This section introduces the technologies supporting the work in this thesis. The first is a platform for software development. The platform’s tools provide robust processing capabilities for the Java programming language. The second is a relational manipulation language, and its interpreter, for the relational analysis of software.

2.4.1 Eclipse

Eclipse [12] is an open development platform supporting several programming languages. It supports a plug-in architecture that allows developers to modify the platform through third-party plugin-ins, which they can implement themselves. Primarily, though, it has been distributed as an integrated development environment for Java. At its core is Java Development Tools (JDT) [13], a subproject of Eclipse that provides an infrastructure for Java development.

Java Development Tools

JDT provides a compiler, incremental builder, and debugger for the Java programming language. The compiler infrastructure, in particular, can be leveraged for the
static analysis of Java source code. The libraries can be used to parse compilation units and resolve bindings for source code elements. This kind of information is useful for constructing the dependence graphs on which object-oriented, static analyses are based.

Parsing and manipulation of Java programs are implemented by the classes contained in the packages: \texttt{org.eclipse.jdt.core} and \texttt{org.eclipse.jdt.core.dom} (Figure 2.4 on the next page). The first package contains classes for representing Java programs in terms of their source code elements. These elements are referred to as the Java program model. The second package contains classes for parsing Java programs to generate abstract syntax trees. These classes also include the ability to resolve bindings for abstract syntax tree nodes that contain references to source code elements.

The source code elements comprising the Java program model include: compilation units, types, fields, methods, and local variables. A compilation unit, represented by the interface \texttt{ICompilationUnit}, is parsed using an \texttt{ASTParser}. The parse generates an abstract syntax tree whose nodes are represented by the abstract class \texttt{ASTNode}. Nodes containing references to source code elements may have their references resolved to bindings, which are represented by the interface \texttt{IBinding}. In turn, a binding has a source code element to which it is bound. For example, if a method \(m\) of a class \(T\) is invoked somewhere in a compilation unit \(C\), parsing \(C\) generates an abstract syntax tree with a subtree for the method invocation. The nodes in this subtree are bound to an \texttt{IType} representing \(T\), and an \texttt{IMethod} representing \(m\).

An \texttt{ASTVisitor} is used to perform operations on an abstract syntax tree. It traverses the tree starting from the root node, and recursively accesses the nodes in
Figure 2.4: Class diagram of the core components in JDT.
each of the subtrees. As the nodes are accessed, their properties can be examined to extract information about the source code elements and their relationships.

### 2.4.2 Relational Manipulation

Relations and graphs are used extensively for the analysis of software [5]. With these kinds of analyses, relationships among source code elements are represented as relations. The relations may then be manipulated to form higher-level ones, or to infer new ones. The Relational Manipulation Language (RML) and its interpreter, CrocoPat, were created to support relational analysis both generally and efficiently [5]. They are general because they support relations of arbitrary arity. They are efficient because they use a highly compressible data structure to represent the relations.

Programs written in RML are used to manipulate and query relations. The language is based on first-order predicate calculus, but also includes control structures like branches and loops. An RML program consists of a sequence of statements, of which there are several types, including: assignments, and control.

An assignment statement assigns the result of a relational expression to a relation variable:

\[
\text{lessThan}(m,n) := \text{greaterThan}(n,m);
\]

Here, the binary relation variable \textit{lessThan} is assigned the inverse of the binary relation \textit{greaterThan}. More complex relational expressions can be created using the RML operators for complement, intersection, and union: \(\neg\), \&, \(|
\]

\[
\text{lessThanOrEqualTo}(m,n) := \text{lessThan}(m,n) \cup \text{equals}(m,n);
\]

Additionally, terms can be existentially or universally quantified using \texttt{EX} or \texttt{FA}:
Chapter 2. Background

\[
\begin{align*}
\text{child}(t) & := \text{EX}(u, \text{parent}(u,t)) \\
\text{child}(t) & := \text{parent}(\_ , t)
\end{align*}
\]

Here, the relation variable \textit{child} is assigned the set of elements for which there exists at least one \textit{parent}. The second statement is equivalent to the first, but has a more compact syntax.

The control statements are similar to those for other imperative languages. There are loops (e.g., FOR, WHILE) and branches (e.g., IF-ELSE).

2.5 Summary

This chapter reviewed some related work in the area of reverse engineering, and provided information that is helpful for understanding the concepts discussed in later chapters. In the related work, there is a common theme of analyzing source code elements, such as variables and control statements, and their relationships. These analyses attempt to infer the conceptual roles of variables.

As outlined in the next chapter, a similar approach is taken in this thesis. In this case, the objective is to identify variables that fit the role of a tuning parameter.
Chapter 3

Overview

This chapter provides an overview of the approach taken to automate the recovery of software tuning parameters. The approach has four phases: empirical study, fact extraction, fact manipulation, and pattern matching. More detail about the phases is provided in later chapters.

The empirical study is conducted initially to determine how tuning parameters are manifested in source code. The tuning parameters are gathered from documentation and mapped to field declarations. References to the fields are then traced to observe how they are accessed. Characteristics of these references are used to develop patterns for identifying other, similar fields.

A metamodel of the source code element relationships needed to express the patterns is produced. Based on the metamodel, an extractor is developed to automatically obtain instances of the relationships from source code. Object-oriented languages introduce indirection through method invocations and polymorphism. The extracted relationships are therefore manipulated to create new ones between the indirectly related elements. After the new relationships have been generated, they are
analyzed for instances of the patterns elicited from the study. The result is a set of potential tuning parameters.

When developing a new pattern, the potential tuning parameters are validated by comparing them to the set of documented tuning parameters. The metamodel and patterns are refined iteratively, as needed, to improve accuracy. Once the patterns and metamodel have stabilized, the process can be applied to other applications without the empirical study step, and instead starting with the fact extraction.

The processes and artifacts constituting the approach are depicted in Figure 3.1. The rectangles represent automated processes, the trapezoid represents a manual process, and the curved bottom rectangles represent input and output artifacts. Descriptions of the processes and artifacts are given in the following sections.
3.1 Empirical Study

The empirical study is a manual process taking tuning parameters and source code as input, to produce a metamodel and a set of patterns as output. Initially, the tuning parameters are taken from an application’s documentation. By mapping the documented tuning parameters to field declarations in the application’s source code it is possible to observe how they are manifested. The similarities and differences that exist in how the fields are referenced form the basis for the patterns. In addition, the features of each pattern dictate what source code elements and relationships are needed to identify it. These elements and relationships are used to develop the metamodel.

Once an iteration of the approach is completed, the patterns are validated by comparing the resulting set of recovered tuning parameters to the known set. If there are too many false-negatives or false-positives, the source code is re-examined, and the patterns and metamodel are refined. When an acceptable accuracy is achieved, the automated phases of the approach may be applied to other applications for which the tuning parameters are initially unknown. As indicated in Figure 3.1 on the previous page, these potential tuning parameters may be validated by empirical study of the application. This validation may lead to further refinement of the metamodel and patterns.

3.2 Fact Extraction

The automated extraction process takes the metamodel and source code as input, to produce a strong relationship graph as output. The metamodel developed during the
empirical study is transformed into a graph schema that is implemented as custom logic for extracting the elements and relationships from source code. The extraction is per compilation unit.

As shown in Figure 3.2, the process begins by parsing the compilation unit. This parse generates an abstract syntax tree and, where applicable, resolves any references in the tree’s nodes. The tree is then traversed to extract relationship instances, called facts. A set of facts corresponding to a particular relationship type can be thought of as a binary relation over the element types involved in that relationship. As depicted in Figure 3.2, these relations are represented as a directed graph of source code element nodes and relationship edges. The graph is referred to as being “strong” since the relationships it represents are taken directly from the source code.
3.3  Fact Manipulation

The automated manipulation process takes the strong relationship graph as input, to produce a weak relationship graph as output. The output graph is referred to as being “weak” since it represents an over-approximation of the indirect relationships inferred from the strong relationship graph. For instance, the formal parameters of a method’s declaration would have relationships generated for all possible actual parameters of that method’s invocations, including invocations it may override. This manipulation process helps to overcome the indirection induced by the object-oriented paradigm. In this way, the per-compilation unit facts from the extraction process are combined to create a holistic representation of the program. In addition, certain low-level relationships are abstracted to create higher-level ones. While not strictly necessary, these manipulations make it easier to express the patterns for matching in the next phase.

As indicated in Figure 3.3 on the next page, the manipulations are expressed in a language for querying and manipulating relations that is based on first-order predicate calculus. This language was created for the CrocoPat [5] relational programming tool. While the implementation does not use CrocoPat for reasons of practicality, use of the language will help with a possible integration in the future.

3.4  Pattern Matching

The automated pattern matching process takes the weak relationship graph and tuning parameter patterns as input, to produce a set of potential tuning parameters as output. A tuning parameter pattern is specified as a relational expression that,
Figure 3.3: Manipulating the graph to create new relationships.

Figure 3.4: Pattern matching to identify tuning parameters.
when applied to the weak relationship graph, generates a new relation whose mem-
bbers satisfy the constraints of that expression. The relational expression describes
the relationships that are common to all tuning parameters of a kind, as well as those
relationships that differentiate that kind from others.

The pattern matching is a kind of relation, or graph, query and is therefore related
to the subgraph isomorphism problem. In graph theory terms, the problem is to
identify subgraphs of the weak relationship graph that correspond to tuning parameter
use. For example, the subgraph with bold edges depicted in Figure 3.4 on the previous
page would represent such a match.

3.5 Summary

The automation of tuning parameter recovery has been divided into four phases: em-
pirical study, fact extraction, fact manipulation, and pattern matching. This chapter
provided a brief introduction to these phases, and related the artifacts consumed and
produced by each. The next four chapters are dedicated to discussing these phases
in more detail, and providing examples to illustrate their implementation.
Chapter 4

Empirical Study

Chapter 3 introduced the four phases of automating software tuning parameter recovery. The first of these phases, empirical study, involves observing how tuning parameters are manifested in source code. In Chapter 1, a tuning parameter was defined as a scalar field explicitly declared in source code that measures or controls non-functional factors such as performance. It is therefore important to understand how tuning parameters are manifested as fields in source code.

In this chapter, fields corresponding to tuning parameters are studied to elicit patterns in how they are referenced. As such, the tuning parameters are organized into a taxonomy based on these patterns. The study looks at four applications. Two are chosen from the web application domain, and two are chosen from the database application domain. This distinction is made to avoid bias toward either the applications, or application domains. Table 4.1 on the next page provides an overview of the applications, which were chosen based on the following criteria:

- Must be open source;
- Must be implemented in the Java programming language;
CHAPTER 4. EMPIRICAL STUDY

Table 4.1: Overview of the applications studied.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Domain</th>
<th>Source</th>
<th>License</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Derby</td>
<td>10.2.2.0</td>
<td>Database</td>
<td>Apache 2.0</td>
<td>Apache 2.0</td>
<td>425,366</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>6.0.10</td>
<td>Web/Servlet</td>
<td>Apache 2.0</td>
<td></td>
<td>125,366</td>
</tr>
<tr>
<td>Oracle Berkeley DB</td>
<td>3.2.23</td>
<td>Database</td>
<td>Oracle (OSI Approved)</td>
<td></td>
<td>67,842</td>
</tr>
<tr>
<td>Mort Bay Jetty</td>
<td>6.1.3</td>
<td>Web/Servlet</td>
<td>Apache 2.0</td>
<td></td>
<td>35,723</td>
</tr>
</tbody>
</table>

- Should be server-oriented;

- Should have some industrial relevance.

The first requirement is motivated by the study itself, which involves observing fields in source code. The second requirement is motivated by prior work on a STAC prototype from [9] that works with Java. The last two guidelines emphasize tuning of server-oriented applications—as opposed to client-oriented—when such applications are deployed in industrial environments. Self-management is becoming increasingly important in this context.

The initial task is to gather the applications’ documented tuning parameters. An application’s documentation usually describes its tuning parameters so the administrator can know what they are, and what they measure or control. As well, the Java programming language has a standard for implementing management components. Components that adhere to the Java Management Extensions (JMX) specification follow well-known naming conventions and design patterns. Naming conventions enable introspection of a component’s properties since each property’s accessor and mutator method names use a standard prefix. These conventions are not specific to management, but to properties of Java objects in general. Both sources of information are used to gather the set of tuning parameters for an application.
Once the tuning parameters are gathered, they are mapped to field declarations. The mapping process involves syntactically searching for fields whose names match exactly, or are similar to, the names of tuning parameters. There are three possible outcomes:

1. Exactly one matching declaration is found;
2. More than one matching declaration is found;
3. No matching declarations are found.

The first outcome is the most common, and is quickly validated by inspection. The second outcome is less frequent than the first, though more common than the third. The additional matches in this case are often the result of data transfer objects. These objects act as containers in object-oriented languages to transfer a set of values as a single argument to a method invocation, or as a single return value from a method invocation. The data transfer class declares fields whose identifiers resemble those of the fields that are being transferred. Matches of this type contribute nothing to the mapping. Additional matches also result when a tuning parameter with the same names exists for more than one component. These matches represent separate instances that happen to have the same identifier. As with the first, this case can be quickly validated by inspection. The last case is the most troublesome since either the tuning parameter is deprecated, or it has been named differently in the source code, or it has been implemented indirectly as a hashed key/value pair. The former cases are indistinguishable without additional clues and a more in-depth validation. The latter case is confirmed by weakening the search to be lexical.

Having mapped the tuning parameters to field declarations, all references to the fields are traced. The references are studied to determine how the tuning parameters
are manifested in the source code. The question being addressed is whether patterns emerge that can be used to identify them, as well as others of the same kind.

The next section presents the applications and their tuning parameters. It outlines how the tuning parameters are gathered for each application, and how they are mapped to field declarations. A complete catalogue of the mappings is provided in Appendix A. The section after that provides examples of stereotypical tuning parameter use. These serve as a starting point for classifying the tuning parameters, and for the formal expression of patterns in Chapter 7.

4.1 Applications

4.1.1 Apache Tomcat

Apache Tomcat is a servlet container providing server-side dynamic web content using the Java Servlet technology. It can be configured as a backend to an existing web server, or can be deployed stand-alone using the integrated HTTP server. It serves as the reference implementation for the JavaServer Pages and Java Servlet technologies, and is used in production by hundreds of thousands of websites [25]. Its source code is maintained by the Apache Software Foundation, where it has been developed as an open source project for almost a decade.

Tomcat provides a comprehensive set of JMX components, or managed beans, for its management infrastructure. However, the documentation does not explicitly mention each of the managed beans or their capabilities. Instead, descriptions of them are provided in XML files. The descriptions contain detailed information about the managed beans, including their attributes and operations. Only the scalar attributes
are considered potential tuning parameters for the study.

The description format is specified by a Document Type Definition, or DTD. A DTD defines what elements and element properties are legal in an XML document type. XML documents are composed of elements delimited by tags, which have optional properties in the form of name/value pairs. Managed beans descriptions are composed of zero or more mbean elements, each having zero or more attribute elements and operation elements. An mbean element has a type property that is the fully qualified name of the underlying Java class being managed. An attribute element has a name property and a type property. The name property identifies the attribute according to managed bean naming conventions. The type property contains the fully qualified name of the attribute’s data type.

Listing 4.1 on the next page is an example of a managed bean description from the package org.apache.catalina.connector. Lines 3–8 describe a bean named CoyoteConnector that manages instances of the class Connector. Lines 10–12 describe a scalar attribute named acceptCount that controls the capacity of the server connection queue. The acceptCount attribute is considered a potential tuning parameter.

An XML parser is used to generate a list of the mbean and attribute elements. The type property of the mbean element directs the search to a particular class, while the attribute name directs it to a particular field of that class. From the example, the class Connector in the package org.apache.catalina.connector would be searched for the field acceptCount. A syntactical search for the field yields no matches. But a lexical search for the term “acceptCount” returns the line:

```java
replacements.put("acceptCount", "backlog");
```
Listing 4.1 Description of a managed bean from Apache Tomcat.

```xml
<?xml version="1.0"?>
<mbeans-descriptors>
  <mbean name="CoyoteConnector"
    className="org.apache.catalina.mbeans.ConnectorMBean"
    description="Implementation of a Coyote connector"
    domain="Catalina"
    group="Connector"
    type="org.apache.catalina.connector.Connector">
    ...
    <attribute name="acceptCount"
      description="The accept count for this Connector"
      type="int"/>
    ...
  </mbean>
</mbeans-descriptors>
```

Using this match as a hint, the search continues for a field named `backlog`. Again, the search does not find a match. The search is then widened to other classes. This time matches in several different classes are found. Upon inspection, these classes are referenced by the `Connector` class, which indirectly sets the value of `backlog` using Java’s reflection mechanism. Satisfied that the matches are valid, the `acceptCount` tuning parameter is mapped to each of the declarations. This process is followed for the remaining managed beans and attributes to create the other mappings.

### 4.1.2 Mort Bay Jetty

Mort Bay Jetty is a web server and Java servlet container. It can be deployed stand-alone to provide static and dynamic web content, though it was designed mostly to be embedded into other applications. It is the default servlet container for several Apache projects, and is also used in the Eclipse development environment. It was first released in 1995 as a project maintained by Mort Bay Consulting under the same
open source license as Apache Software Foundation projects like Tomcat.

Mort Bay Jetty, like Apache Tomcat, provides a comprehensive set of managed beans. However, rather than using XML to describe them, the managed beans in Jetty are described using an informal, line-oriented format. As shown in Listing 4.2, the format is simple enough that a grammar can be deduced. A line contains either a managed bean name, attribute name, or operation name, along with a human-readable description. If the name begins with an uppercase letter, it is a bean. If the name begins with a lowercase letter, it is an attribute or operation. Operations can be distinguished from attributes by the suffix "()". The names are separated from the descriptions by a colon.

Using the deduced grammar, the descriptions are parsed to produce a list of potential tuning parameters. The managed bean name directs the search to a particular class, while the attribute name directs it to a particular method of that class. The naming conventions for managed beans dictate that the bean’s class have an accessor method based on the attribute’s name. The goal is to find the field that this method accesses. In Listing 4.2, line 1 describes a managed bean named \texttt{StatisticsHandler}, while line 4 describes an attribute named \texttt{requests}. This attribute is considered a potential tuning parameter.

Searching for a class corresponding to the managed bean leads to the package \texttt{org.mortbay.jetty.handler}. Within the class is a method named \texttt{getRequests()}

\begin{lstlisting}[language=java]
1 StatisticsHandler: Request Statistics gathering
2 ...
3 statsReset(): Reset statistics.
4 requests: Number of requests since statsReset() called.
5 ...
\end{lstlisting}
that accesses a field named _requests:

```java
package org.mortbay.jetty.handler;
...
public class StatisticsHandler extends HandlerWrapper {
    ...
    transient int _requests;
    ...
    public int getRequests()
    {
        return _requests;
    }
    ...
}
```

The match is consistent with the managed bean description, so the requests tuning parameter is mapped to the _requests field. This process is followed for the remaining managed beans and attributes to create the other mappings.

### 4.1.3 Apache Derby

Apache Derby is a relational database management system. It can be embedded into an application, or deployed in a more traditional client/server network architecture. It was originally developed by Cloudscape Inc., before being acquired by IBM, who eventually donated the source code to the Apache Software Foundation in 2004. Sun Microsystems Inc. has recently started distributing a supported version of Derby with its Java development toolkits.

Unlike the other applications in this study, Apache Derby does not have a JMX implementation, though preliminary support is scheduled for release in version 10.4. This means there is no formal description of its management capabilities. Thus far, the JMX components have included two kinds of attributes: those that control
behaviour, and those that measure it. While the former are useful for changing behaviour, the latter are important for knowing where the changes should be made. Unfortunately, the documentation for Derby does not describe these, which limits the kinds of tuning parameters that can be gathered.

The tuning parameters for the embedded database engine are described in [3]. Those for the network server are described in [2]. Both manuals use a similar naming convention for the parameters. The names are dot separated, and include the application name, followed by a component name, followed by the actual parameter name. For example, `derby.locks.monitor`. Once the parameters are gathered from the manuals, the source code is searched lexically for the parameter names. As one might expect, they are all located in an interface called `Property` that declares fields that are assigned the parameter names:

```java
package org.apache.derby.iapi.reference;
...
public interface Property
{
    ...
    public static final String DEADLOCK_MONITOR = "derby.locks.monitor";
    ...
}
```

These are used as constants in the source code to refer to the parameters. Mapping the parameters to fields involves tracing the references to these constants.

### 4.1.4 Oracle Berkeley DB Java Edition

Oracle Berkeley DB Java Edition (JE) is an embedded storage engine. Unlike more traditional database management systems that have a client/server network architecture, JE resides in the application’s address space and provides transactional storage
directly using an application programming interface. The name Berkeley DB was originally given to the C version of the storage engine that was developed at the University of California at Berkeley and then later distributed by Sleepycat Software, Inc. Oracle acquired the company in 2006 and now distributes both the C and Java editions. The source code is available under an Oracle license that has been approved by the Open Source Initiative.

JE, like Apache Tomcat and Mort Bay Jetty, provides a JMX implementation. However, managed beans in JE are described directly in the source code using the JMX management interfaces. Listing 4.3 on the next page shows the managed bean that acts as a facade for all the management capabilities of JE. This one managed bean contains the attributes and operations supported by the implementation.

Lines 8–13 describe an attribute for the database’s cache size, while lines 18–22 describe an operation for retrieving statistics about table and row locks. Recall that managed bean operations were ignored in the previous applications. In this case, the operations need to be considered. The difference is due to the granularity of the operations. For the other applications, an operation was a specific task that invoked some behaviour. These did not have corresponding fields to which they could be mapped. In JE, the operations are coarse. For example, the operation in Listing 4.3 on the next page populates an object with data—the data being locking statistics. The object in this case is an instance of the class `com.sleepycat.je.LockStats`. These statistics would be useful tuning parameters. Ignoring the operation could mean missing potential tuning parameters. For operations like this one, the fields of the object are considered tuning parameters. Though they are themselves fields, they are still distinct from the actual manifestations. They merely act as holders for the
Listing 4.3 Description of a managed bean from Oracle Berkeley DB Java Edition.

```java
package com.sleepycat.je.jmx;
...
public class JEMBeanHelper
{
    private static final MBeanAttributeInfo[] OPEN_ATTR =
    {
        ...
        new MBeanAttributeInfo(ATT_CACHE_SIZE,
            "java.lang.Long",
            "Cache size, in bytes.",
            true, // readable
            true, // writable
            false), // isIs
        ...
    }
    ...
    private static final MBeanOperationInfo OP_LOCK_STAT_INFO =
    new MBeanOperationInfo(OP_LOCK_STAT,
        "Get locking statistics.",
        statParams,
        "com.sleepycat.je.LockStats",
        MBeanOperationInfo.INFO);
    ...
}
```
transference of data. The mappings for these tuning parameters are generated using reference tracing, instead of syntactical or lexical searching.

4.2 Patterns

Now that the tuning parameters are mapped to field declarations, all references to the fields are traced to observe how the fields are used. This is done in a recursive, depth-first manner. A field reference is itself traced for references, and so on, until no more are found. The references indicate all of the ways in which a field is accessed, directly or indirectly, in the source code. The fields are then grouped according to characteristics shared by their references. For example, certain fields are only ever incremented using the Java increment operator, \texttt{++}. This fact usually indicates that the field is used to count occurrences of an event. Fields having this characteristic are classified as \textit{counters}.

Classifications developed from the four applications are shown in Figure 4.1 on the next page. The next few sections describe the classifications, and provide examples to illustrate their characteristics. In Chapter 7, several of the patterns are expressed more formally so they can be used by the automated pattern matching process.

4.2.1 Behavioural

The behavioural classification is given to tuning parameters that control an application’s behaviour using some finite set of constant values. It has two subclassifications: toggle, and enumerable.
Figure 4.1: Classification of tuning parameter patterns.
Toggle

A toggle uses two values. The controlled behaviour is enabled or disabled. The toggle pattern can be characterized as a field that is evaluated in the condition of an if statement or ?: (conditional) operator. The evaluation may involve subexpressions of conditional operators such as: 11, &&, and !. Figure 4.2 is an example of a toggle named disableUploadTimeout. It controls whether timeouts are employed when reading from a socket.

<table>
<thead>
<tr>
<th>Toggle</th>
<th>Enables or disables a behavior of an application.</th>
</tr>
</thead>
</table>
|         |     ...  
|         |     public class Http11Processor implements ActionHook {  
|         |     ...  
|         |     protected boolean disableUploadTimeout = false;  
|         |     ...  
|         |     public void process(Socket socket)  
|         |         throws IOException {  
|         |     ...  
|         |     if (!disableUploadTimeout && keptAlive) {  
|         |         if (keepAliveTimeout > 0) {  
|         |             socket.setSoTimeout(keepAliveTimeout);  
|         |         }  
|         |         else if (soTimeout > 0) {  
|         |             socket.setSoTimeout(soTimeout);  
|         |         }  
|         |     }  
|         |     ...  
|         |     if (!disableUploadTimeout) {  
|         |         socket.setSoTimeout(timeout);  
|         |     }  
|         |     ...  
|         |     }  
|         | }  
|         | }  

Figure 4.2: Stereotype of the toggle classification.
Enumerable

An enumerable uses two or more values. The controlled behaviour is determined by the enumerable’s value. The enumerable pattern can be characterized as a field that is evaluated in contexts where its equality to some constant value determines which of two or more execution paths are taken. Figure 4.3 on the next page is an example of an enumerable named report. It controls how exceptions are reported. The evaluation context in this case is a switch statement.
Enumerable  Controls the behavior of an application using some finite set of constant values.

Example - Apache Derby

```java
package org.apache.derby.iapi.error;
...
public class StandardException extends Exception
{
  public static final int REPORT_DEFAULT = 0;
  public static final int REPORT_NEVER = 1;
  public static final int REPORT_ALWAYS = 2;
  ...
  private transient int report;
  ...
  public final int report() {
    return report;
  }
  ...
}
```

```java
package org.apache.derby.iapi.services.context;
...
public class ContextManager {
  ...
  private boolean reportError(Throwable t) {
    ...
    StandardException se = (StandardException) t;
    switch (se.report()) {
      case StandardException.REPORT_DEFAULT:
        ...
      case StandardException.REPORT_NEVER:
        ...
      case StandardException.REPORT_ALWAYS:
        default:
          ...
      }
    ...
  }
  ...
}
```

Figure 4.3: Stereotype of the enumerable classification.
4.2.2 Numerical

The numerical classification is given to those tuning parameters that measure or control an application’s behaviour using numeric values. It has four subclassifications: accumulator, capacity, counter, and threshold.

Accumulator

An accumulator maintains the algebraic sum of a variable, or variables, over time. The accumulator pattern can be characterized as a field that is incremented by some variable. The incrementation may be expressed as an implicit self-assignment using the $+=$ operator, or as an explicit self-assignment using the $=$ operator and an infix expression with the $+$ operator between the field and the variable. Figure 4.4 on the next page is an example of an accumulator named nLMQueueHits. It maintains the cumulative total of successful accesses of a queue.
**Accumulator** Measures how many times an event or action occurs in an application as discrete, variable increments.

**Example - Oracle Berkeley DB Java Edition**

```java
package com.sleepycat.je.cleaner;
...
public class Cleaner implements DaemonRunner, EnvConfigObserver {
  ...
  int nLNQueueHits = 0;
  ...
}
```

```java
package com.sleepycat.je.cleaner;
...
class FileProcessor extends DaemonThread {
  ...
  private void accumulatePerRunCounters() {
    ...
    cleaner.nLNQueueHits += nLNQueueHitsThisRun;
    ...
  }
  ...
}
```

Figure 4.4: Stereotype of the accumulator classification.

**Capacity**

A capacity limits how large a resource can grow. Generally, this resource is a container whose capacity limits the number of items it can hold. The capacity pattern can be characterized as a field that is evaluated in contexts that determine the dimension of an array. Figure 4.5 on the next page is an example of a capacity named `logBufferSize`. It controls the number of bytes in a buffer.
Capacity

Controls the number of items a container in an application can hold.

Example - Apache Derby

```java
package org.apache.derby.impl.store.raw.log;
...
public final class LogToFile implements LogFactory, ...
{
  ...
  ... StandardException
  {
    ...
    logOut = new LogAccessFile(this, firstLog, logBufferSize);
    ...
  }
  ...
}
```

```java
package org.apache.derby.impl.store.raw.log;
...
public class LogAccessFile 
{
  ...
  public ...  int bufferSize) 
  {
    ...
    LogAccessFileBuffer b = new LogAccessFileBuffer(bufferSize);
    ...
  }
  ...
}
```

```java
package org.apache.derby.impl.store.raw.log;
...
final class LogAccessFileBuffer
{
  ...
  public LogAccessFileBuffer(int size)
  {
    buffer = new byte[size];
    ...
  }
  ...
}
```

Figure 4.5: Stereotype of the capacity classification.
Counter

A counter is similar to an accumulator, except that it maintains the algebraic sum, difference, or both, of some constant value over time. The counter pattern can be characterized as a field that is incremented, decremented, or both. Figure 4.6 is an example of a counter named `nCacheMiss`. It maintains a count of unsuccessful accesses of a cache. The example counter is only ever incremented, so it would be given the monotonically increasing subclassification. Conversely, a counter that is only ever decremented would be given the monotonically decreasing subclassification. A counter that is both incremented and decremented is said to be nonmonotonic.

<table>
<thead>
<tr>
<th>Counter</th>
<th>Measures how many times an event or action occurs in an application as discrete, constant increments, decrements, or both.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example - Oracle Berkeley DB Java Edition</strong></td>
<td></td>
</tr>
<tr>
<td>package com.sleepycat.je.log;</td>
<td></td>
</tr>
</tbody>
</table>
|  | class LogBufferPool {
|  |  |  | ...  |
|  |  |  | private long nCacheMiss = 0; |
|  |  |  | } |
|  |  |  | LogBuffer getReadBuffer(long lsn) throws DatabaseException {
|  |  |  |  | ... |
|  |  |  |  |  | if (foundBuffer == null) {
|  |  |  |  |  | nCacheMiss++; |
|  |  |  |  |  | } |
|  |  |  |  | } |
|  |  |  |  | } |

Figure 4.6: Stereotype of the counter classification.
Threshold

A threshold is the magnitude at which some numerical value exceeds its desired range. The threshold pattern can be characterized as a field that is compared to an expression in the condition of a control statement. The comparison is performed using one of the relational operators: $>$, $<$, $\geq$, $\leq$. Figure 4.7 on the next page is an example of a threshold named `minimumRecordSize`. It controls the minimum space needed to store a database record. A threshold that triggers a behaviour when the value is below the threshold, like the one in the example, would be given the lowerbound subclassification. Conversely, a threshold that triggers a behaviour when the value is above the threshold would be given the upperbound subclassification.
Threshold Controls the magnitude at which some numerical value triggers an action or event in an application.

Example - Apache Derby

```java
package org.apache.derby.impl.store.raw.data;
...
public class StoredPage extends CachedPage {
  ...
  protected int minimumRecordSize;
  ...
  public boolean allowInsert() {
    ...
    if (spaceAvailable < minimumRecordSize)
      return false;
    ...
  }
  ...
  protected boolean spaceForCopy(int spaceNeeded) {
    int bytesNeeded = slotEntrySize +
      (spaceNeeded >= minimumRecordSize ? spaceNeeded : minimumRecordSize);
    return((freeSpace - bytesNeeded) >= 0);
  }
  ...
}
```

Figure 4.7: Stereotype of the threshold classification.

4.2.3 Statistical

The statistical classification is given to those tuning parameters that measure variability in a numerical or temporal value over time. It has three subclassifications: average, maximum, and minimum.
Average

An average maintains the statistical mean of a numerical or temporal value over time. The average pattern can be characterized as a field that is assigned an expression that computes a statistical mean. The general form of statistical mean computes the sum a group of values, then divides the result by the group’s cardinality. Figure 4.8 on the next page is an example of an average named `sessionAverageAliveTime`. It maintains the statistical mean of the duration of all client sessions with the server. The mean in this example is computed iteratively by first computing the previous sum, adding the new value, then dividing by the previous cardinality plus one.
Average  Measures the statistical mean of a numerical or temporal value over time.

Example - Apache Tomcat

```java
package org.apache.catalina.session;
...
public abstract class ManagerBase implements Manager, MBeanRegistration
{
    ...
    protected int sessionAverageAliveTime;
    ...
    public int getSessionAverageAliveTime() {
        return sessionAverageAliveTime;
    }
    ...
    public void setSessionAverageAliveTime(int sessionAverageAliveTime) {
        this.sessionAverageAliveTime = sessionAverageAliveTime;
    }
    ...
}
```

```java
package org.apache.catalina.session;
...
public class StandardSession implements HttpSession, Session, Serializable
{
    ...
    public void expire(boolean notify)
    {
        ...
        int average = manager.getSessionAverageAliveTime();
        average = ((average * (numExpired-1)) + timeAlive)/numExpired;
        manager.setSessionAverageAliveTime(average);
    }
    ...
}
```

Figure 4.8: Stereotype of the average classification.
Maximum

A maximum maintains the statistical maximum of a numerical or temporal value over time. The maximum pattern can be characterized as a field that is only assigned an expression to which it has been evaluated as being greater than. Figure 4.9 on the next page is an example of a maximum named `sessionMaxAliveTime`. It maintains the statistical maximum of the duration of all client sessions with the server.
**Maximum** Measures the statistical maximum of a numerical or temporal value over time.

**Example - Apache Tomcat**

```java
package org.apache.catalina.session;
...
public abstract class ManagerBase implements Manager, MBeanRegistration {
  ...
  protected int sessionMaxAliveTime;
  ...
  public int getSessionMaxAliveTime() {
    return sessionMaxAliveTime;
  }
  ...
  public void setSessionMaxAliveTime(int sessionMaxAliveTime) {
    this.sessionMaxAliveTime = sessionMaxAliveTime;
  }
  ...
}
```

```java
package org.apache.catalina.session;
...
public class StandardSession implements HttpSession, Session, Serializable {
  ...
  public void expire(boolean notify) {
    ...
    long timeNow = System.currentTimeMillis();
    int timeAlive = (int) ((timeNow - creationTime)/1000);
    ...
    if (timeAlive > manager.getSessionMaxAliveTime()) {
      manager.setSessionMaxAliveTime(timeAlive);
    }
  }
  ...
}
```

Figure 4.9: Stereotype of the maximum classification.
Minimum

A minimum maintains the statistical minimum of a numerical or temporal value over time. The minimum pattern can be characterized as a field that is only assigned an expression to which it has been evaluated as being less than. Figure 4.10 is an example of a minimum named \_requestActiveMin. It maintains the statistical minimum of the number of requests actively being handled by the server.

<table>
<thead>
<tr>
<th>Minimum</th>
<th>Measures the statistical minimum of a numerical or temporal value over time.</th>
</tr>
</thead>
</table>
| Example - Mort Bay Jetty | package org.mortbay.jetty.handler;  
...  
public class StatisticsHandler extends HandlerWrapper  
{  
...  
  transient int _requestsActiveMin;  
...  
  public void handle(  
    String target, HttpServletRequest request,  
    HttpServletResponse response, int dispatch)  
  throws IOException, ServletException  
  {  
...  
    if (_requestsActive < _requestsActiveMin)  
      _requestsActiveMin=_requestsActive;  
...  
  }  
...  
} |

Figure 4.10: Stereotype of the minimum classification.
4.2.4 Temporal

The temporal classification is given to those tuning parameters that measure or control an application’s behaviour using relative or absolute amounts of time. It has five subclassifications: delay, duration, period, point, and timeout.

Delay

A delay prevents a behaviour from occurring for some specified length of time. The delay is usually accomplished by causing the current thread of execution to leave the runnable state. The delay pattern is therefore characterized as a field that is evaluated in contexts that cause a thread to wait for some time interval. Figure 4.11 on the next page is an example of a delay named `backgroundSleepInterval` that causes a thread to wait using the Java API method `Thread.sleep()`.
**Delay** Controls the length of time before an action or event in an application is triggered.

**Example - Oracle Berkeley DB Java Edition**

```java
package com.sleepycat.je.dbi;
...
public class EnvironmentImpl implements EnvConfigObserver {
    ...
    private long backgroundSleepInterval;
    ...
    public void sleepAfterBackgroundIO() {
        if (backgroundSleepBacklog > 0) {
            ...
            Thread.sleep(backgroundSleepInterval);
            ...
        }
    }
    ...
}
```

Figure 4.11: Stereotype of the delay classification.

**Duration**

A duration maintains the cumulative time spent performing a behaviour. The duration pattern can be characterized as a field that is incremented by the difference between two temporal points; a temporal point is discussed later in this section. Figure 4.12 on the next page is an example of a duration named `_connectionsDurationTotal`. It maintains the total time clients spent connected to the server.
**Duration** Measures the length of time an action or event in an application has taken.

**Example - Mort Bay Jetty**

```java
package org.mortbay.jetty;
...
public abstract class AbstractConnector
    extends AbstractBuffers implements Connector
{
...  
    transient long _connectionsDurationTotal;
...
    protected void connectionClosed(HttpConnection connection)
    {
        ...
        long duration=System.currentTimeMillis()-connection.getTimeStamp();
        ...
        _connectionsDurationTotal+=duration;
        ...
    }
...}
```

Figure 4.12: Stereotype of the duration classification.

**Period**

A period determines how frequently a behaviour is repeated. The period pattern can be characterized as a field that is used to increment a temporal point. Figure 4.13 on the next page shows an example of a period named `scanInterval`. It is used as a parameter to a Java API method that schedules tasks for periodic execution. Within the Java library, this parameter’s use is consistent with the period pattern. But the source code for the library is beyond the scope of the application. The parameter itself is therefore also considered a characterization of the period pattern. In other words, a period can be a field that is passed as the third parameter of an invocation
of the method Timer.scheduleAtFixedRate().

<table>
<thead>
<tr>
<th>Period</th>
<th>Controls the frequency with which an action or event in an application is triggered.</th>
</tr>
</thead>
</table>

**Example - Mort Bay Jetty**

```java
package org.mortbay.util;
...
public class Scanner
{
  private int _scanInterval;
  ...
  public int getScanInterval()
  {
    return _scanInterval;
  }
  ...
  public synchronized void setScanInterval(int scanInterval)
  {
    ...
    if (_running && _scanInterval > 0)
      _timer.scheduleAtFixedRate(
        _task,1000L*getScanInterval(),1000L*getScanInterval());
  }
  ...
  public synchronized void start()
  {
    ...
    if (getScanInterval()>0)
      _timer.scheduleAtFixedRate(
        _task,1000L*getScanInterval(),1000L*getScanInterval());
  }
  ...
}
```

Figure 4.13: Stereotype of the period classification.

**Point**

A point maintains the instant in time a behaviour starts or finishes. The point pattern can be characterized as a field that is assigned the current time. Figure 4.14 on the next page is an example of a point named startTime that maintains the time at
CHAPTER 4. EMPIRICAL STUDY

which server contexts are started. The current time is retrieved using the Java API method `System.currentTimeMillis()`.

<table>
<thead>
<tr>
<th>Point</th>
<th>Measures the instant in time an action or event in an application has occurred.</th>
</tr>
</thead>
</table>

**Example - Apache Tomcat**

```java
package org.apache.catalina.core;
...
public class StandardContext
    extends ContainerBase
    implements Context, Serializable, NotificationEmitter
{
    ...
    private long startTime;
    ...
    public synchronized void start() throws LifecycleException
    {
        ...
        startTime = System.currentTimeMillis();
        ...
    }
    ...
}
```

Figure 4.14: Stereotype of the point classification.

**Timeout**

A timeout prevents a behaviour from taking longer than some specified length of time. The time pattern can be characterized as a field that limits a duration. Figure 4.15 on the next page is an example of a timeout named `stateTransferTimeout`. It limits the time taken to propagate state changes to members of a cluster.
CHAPTER 4. EMPIRICAL STUDY

67

Timeout  Controls the length of time an action or event in an application has to complete before being interrupted.

Example - Apache Tomcat

```java
package org.apache.catalina.ha.session;
...
public class DeltaManager extends ClusterManagerBase {
  ...
  private int stateTransferTimeout = 60;
  ...
  public int getStateTransferTimeout() {
    return stateTransferTimeout;
  }
  ...
  protected void waitForSendAllSessions(long beforeSendTime) {
    ...
    do {
      try {
        Thread.sleep(100);
      } catch (Exception sleep) {
        //
      }
      reqNow = System.currentTimeMillis();
      isTimeout = ((reqNow - reqStart) > (1000 * getStateTransferTimeout()));
    } while (!getStateTransfered() && (!isTimeout));
    ...
  }
  ...
}
```

Figure 4.15: Stereotype of the timeout classification.

4.3 Metamodel

The metamodel in Figure 4.16 on the next page specifies the types of source code elements and relationships needed to express the tuning parameter patterns in the previous section. It is a hybrid of middle-level metamodels, like DMM [22], and low-level metamodels, like Datrix [4], used in reverse engineering. To automate the
recovery of tuning parameters, instances of the metamodel are extracted from source code to be analyzed for the patterns.

The metamodel is depicted as a UML class diagram. Boxes depict classes, which represent types of source code elements. Lines depict associations, which represent types of relationships between source code elements. The associations in the metamodel are uni-directional. The direction is depicted as an arrow at one end of the association. Adjacent to the arrow is the association’s multiplicity. The multiplicity specifies the number of class instances allowed to participate in an association instance. For example, the association \textit{isFieldOf} is defined such that a \textit{Field} instance can only be associated with exactly one \textit{Type} instance. There is also a special kind of
association called inheritance. This is depicted as a triangle at one end of the association, near the inherited class. Classes at the other end of the association inherit this class' properties, including its associations. This means, for instance, a Field can be associated with a Method by a returns association.

The classes and associations in the metamodel are described in the next two sections.

### 4.3.1 Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Source code element that can be evaluated. Expressions are com-</td>
</tr>
<tr>
<td></td>
<td>posites that may contain other expressions, and are combined using</td>
</tr>
<tr>
<td></td>
<td>operators.</td>
</tr>
<tr>
<td>Field</td>
<td>Member variable of a class or interface.</td>
</tr>
<tr>
<td>LocalVariable</td>
<td>Local variable of a method or block.</td>
</tr>
<tr>
<td>Method</td>
<td>Method of a class or interface.</td>
</tr>
<tr>
<td>Type</td>
<td>Class or interface.</td>
</tr>
<tr>
<td>Variable</td>
<td>Source code element that contains a value (abstract).</td>
</tr>
</tbody>
</table>
4.3.2 Associations

<table>
<thead>
<tr>
<th>Association</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isIndexOf</td>
<td>Relationship between a subscript, and an array access.</td>
</tr>
<tr>
<td>isConditionOf</td>
<td>Relationship between a condition, and a conditional expression. A conditional statement, such as if, is represented as a collection of conditional expressions.</td>
</tr>
<tr>
<td>isActualParameterOf</td>
<td>Relationship between an actual parameter, and the invoked method (including the parameter’s position).</td>
</tr>
<tr>
<td>isOperandOf</td>
<td>Relationship between an operand, and an expression (including the orientation e.g., left-hand side). This relationship includes: assignments, comparisons, conditional expressions, and arithmetic operations.</td>
</tr>
<tr>
<td>isFieldOf</td>
<td>Relationship between a field, and the type declaring it.</td>
</tr>
<tr>
<td>isFormalParameterOf</td>
<td>Relationship between a formal parameter, and the method declaring it (including the parameter’s position).</td>
</tr>
<tr>
<td>overrides</td>
<td>Relationship between a method, and the interface method it implements or the superclass method it overrides.</td>
</tr>
<tr>
<td>returns</td>
<td>Relationship between a return value, and the method returning it.</td>
</tr>
</tbody>
</table>

4.4 Summary

This chapter looked at the tuning parameters of four industrially relevant, open source applications. The tuning parameters were mapped to field declarations to determine how they were manifested in source code. The direct and indirect references to
the fields were traced to observe how the fields were used. The characteristics of these references were used to organize the tuning parameters into a taxonomy, and to develop informal descriptions of how to recognize the different kinds. The descriptions informally specify patterns of source code elements and relationships that can be used to identify similar ones. The elements and relationships were presented more formally as a metamodel of the information needed from the source code to support the pattern matching process.

Four applications from two application domains were studied. It is uncertain whether the observations from this chapter are more generally applicable. This threat to external validity is addressed somewhat by the evaluation in Chapter 8, which includes two more applications from the original two application domains, as well as a third application from another domain. Also, the selection criteria for the study focused on server-oriented applications. The explicit dichotomy of server-oriented and client-oriented tuning may not exist in practice. For that matter, the terms server-oriented and client-oriented themselves imply a client/server architecture. This implication begs to question whether other system architectures need to be studied as well.

Another issue is the tuning parameter definition presented. The definition has two explicit restrictions: it must be a scalar, and it must be manifested as a field. There are instances where a tuning parameter is not a field, but rather a method that dynamically computes a value. For example, a collection’s size. Properties like this can be useful tuning parameters, but are outside the scope of the study.
Chapter 5

Fact Extraction

The previous chapter described the process of empirically studying tuning parameters to understand how they are manifested as fields in source code. A metamodel of source code elements and relationships was constructed based on the observed patterns of tuning parameter use. In this chapter, an extractor for automatically recognizing the metamodel’s elements and relationships in source code is described. The extractor analyzes the source code to produce strong relationship graphs. The graphs are referred to as being “strong” since they represent relationships taken directly from the source code. This is in contrast to the graphs in the next chapter, which are referred to as being “weak”, since they may represent inferred relationships.

The graphs produced by the extraction are directed graphs. The generic term graph is used throughout this chapter with this definition in mind. A directed graph is a set of nodes and a set of directed edges. A directed edge is an ordered pair, or 2-tuple, of nodes. The pair’s ordering defines the edge’s direction. By definition, the graphs define a set of binary relations. Informally, a binary relation is an association between the elements of a set, or sets. The binary relations correspond roughly to
the associations in the metamodel. They are maintained as a set of tuples, called facts. The set of relations is collectively referred to as a factbase. The terms fact and factbase can be found, to similar effect, in work with the Grok [16] relational calculator. Moreover, Prolog’s [7] use of the terms pre-dates Grok’s. The two use the terms with related, but distinct, meanings. The correspondence between the associations of the metamodel and those of the relations is the topic of the next section.

5.1 Graph Schema

The metamodel in Chapter 4 represented the source code elements and relationships as classes and associations. However, the extraction process produces graphs. The metamodel is therefore transformed into a graph schema. As shown in Figure 5.1 on the next page, the graph schema is itself a graph. The classes and associations from the metamodel are factored, when necessary, to create the graph schema’s nodes and edges. The graphs extracted from the source code are instances of the schema.

A node in the schema is a tuple. This allows it to represent arbitrarily complex source code elements, such as expressions, which are extracted recursively. This means an edge in the schema is a 2-tuple of tuples.
5.1.1 Nodes

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>An Expression node corresponds to the metamodel’s Expression class. It is a $k$-tuple containing a unique identifier for a field, method, or local variable referenced in the source code. Expression nodes can also be combined recursively such that they contain other Expression nodes, and possibly an Operator node.</td>
</tr>
<tr>
<td>Field</td>
<td>A Field node corresponds to the metamodel’s Field class. It is a 1-tuple containing a unique identifier for a field declared in the source code.</td>
</tr>
</tbody>
</table>
### Node Description

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalVariable</td>
<td>A LocalVariable node corresponds to the metamodel’s LocalVariable class. It is a 1-tuple containing a unique identifier for a local variable declared in the source code.</td>
</tr>
<tr>
<td>Method</td>
<td>A Method node corresponds to the metamodel’s Method class. It is a 1-tuple containing a unique identifier for a method declared in the source code.</td>
</tr>
<tr>
<td>Type</td>
<td>A Type node corresponds to the metamodel’s Type class. It is a 1-tuple containing a unique identifier for a type declared in the source code.</td>
</tr>
<tr>
<td>Operator</td>
<td>An Operator node corresponds to the operator property of the metamodel’s Expression class. It is a 1-tuple containing a unique identifier for an assignment, comparison, or arithmetic operator.</td>
</tr>
<tr>
<td>FormalParameter</td>
<td>A FormalParameter node corresponds to the position property of the metamodel’s isFormalParameterOf and isActualParameterOf association classes. It is a 2-tuple containing a unique identifier for a method, and an integer for the parameter’s position.</td>
</tr>
</tbody>
</table>
## 5.1.2 Edges

<table>
<thead>
<tr>
<th>Edge</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isFieldOf</td>
<td>An isFieldOf edge corresponds to the metamodel’s isFieldOf association. It is a 2-tuple containing a Field node and a Type node.</td>
</tr>
<tr>
<td>isFormalParameterOf</td>
<td>An isFormalParameterOf edge corresponds to the metamodel’s isFormalParameterOf association. It is a 2-tuple containing a FormalParameter node and a Method node.</td>
</tr>
<tr>
<td>isActualParameterOf</td>
<td>An isActualParameterOf edge corresponds to the metamodel’s isActualParameterOf association. It is a 2-tuple containing an Expression node and a FormalParameter node.</td>
</tr>
<tr>
<td>isVariableOf</td>
<td>An isVariableOf edge corresponds to the metamodel’s isFormalParameterOf association. It is a 2-tuple containing a LocalVariable node and a FormalParameter node.</td>
</tr>
<tr>
<td>overrides</td>
<td>An overrides edge corresponds to the metamodel’s overrides association. It is a 2-tuple containing Method nodes.</td>
</tr>
<tr>
<td>returns</td>
<td>A returns edge corresponds to the metamodel’s returns association. It is a 2-tuple containing a Method node and an Expression node.</td>
</tr>
<tr>
<td>isOperatorOf</td>
<td>An isOperatorOf edge corresponds to operator property of the metamodel’s Expression class. It is a 2-tuple containing an Operator node and an Expression node.</td>
</tr>
</tbody>
</table>
### 5.2 Source Code

The extraction takes source code as input and extracts the relationships represented in the graph schema. Consider the following source code statement:

\[
x = x + 1;
\]

According to the schema, the graph for this statement has six nodes:

1. (x), a LocalVariable node;
2. (1), an Expression node;
3. (+), an Operator node;
4. (=), an Operator node;
5. \(((x), (((x), (1), (+))), (=))\), an Expression node;

6. \(((x), (1), (+))\), an Expression node;

and six edges:

1. \(((x), ((x), (1), (+))), an isLeftOperandOf edge;

2. \(((1), ((x), (1), (+))), an isRightOperandOf edge;

3. \(((+), ((x), (1), (+))), an isOperatorOf edge;

4. \(((x), ((x), (((x), (1), (+))), (=))), an isLeftOperandOf edge;

5. \(((x), (1), (+)), ((x), (((x), (1), (+))), (=))), an isRightOperandOf edge;

6. \(((=), ((x), (((x), (1), (+))), (=))), an isOperatorOf edge.

The graph is easier to comprehend visually. It looks like this:

This rest of this section describes how Eclipse’s Java Development Tools (JDT) can be leveraged for extracting the graphs from Java source code.

5.2.1 Parser

Source code, prior to processing, is not very different from other textual documents. It contains sequences of characters and symbols. What the characters and symbols
mean is left to interpretation. Parsing, or syntactic analysis, is the process of analyzing
the sequences of characters and symbols in source code to determine their grammatical
structure. A programming language’s grammar defines how the characters and
symbols are grouped, and in what order the groups should appear. The outcome of
syntactic analysis is usually twofold: the implied structure of the text is stored in an
intermediate representation for further processing, and any violations of the grammar
are reported. The extraction process relies on the first outcome, since it is difficult to
recognize relationships between source code elements from just characters and sym-
bols. The second outcome also holds implications for extraction. A parser is usually
best-effort, meaning it continues to work despite grammar violations. This robustness
can result in partial grammatical structures. Relationships involving elements that
have incomplete structures are not extracted.

As a platform for Java development, an integral component of JDT is a parser. The
parser takes source code as input and produces an abstract syntax tree as output. The
abstract syntax tree is an intermediate representation of the source code that holds
its syntactical structure in a set of nodes. Source code elements, such as parentheses
in expressions, can be omitted since the tree implies the structure that these elements
made explicit.

The parser in JDT is represented by the class ASTParser. Abstract syntax tree
nodes are represented by the class ASTNode. These are illustrated by the class diagram
in Figure 5.2 on the next page. All abstract syntax tree nodes are members of a type
hierarchy rooted at ASTNode. Specific kinds of syntactical elements, such as expressions
and statements, are further divided into sub-hierarchies, and so forth.

To generate an abstract syntax tree from source code, a parser instance must be
initialized. As shown in Line 1 of Listing 5.1, the parser instance is created using the 
 newParser() method of the ASTParser class. The grammar employed by the parser is
determined by the edition of the Java Language Specification provided as the parameter. Lines 2–3 provide a handle for the source code to the parser, and enable binding resolution. Bindings are important for linking named source code elements, such as variables, to their references. They provide access to the fully-qualified name for an element, which uniquely identifies that element in an extracted fact. Once initialized, the abstract syntax tree is generated using the method createAST(). The returned ASTNode is the abstract syntax tree’s root.
5.2.2 Visitor

Given an abstract syntax tree, its structure must be traversed to perform operations over the nodes. An operation, in the context of this chapter, is the extraction of source code element relationships. Tree traversal is a common behavioural design problem with a well-established object-oriented solution—the visitor design pattern [14]. A visitor design isolates an algorithm from an object structure, allowing the details of the algorithm to vary independently of the structure. This separation permits novel operations over the structure without having to modify it, or its members. The algorithm for an abstract syntax tree “visits” each of the tree’s nodes. If a node contains a source code element involved in a relationship of interest, that relationship is extracted.

In JDT, an abstract syntax tree visitor is represented by the ASTVisitor class. The association between nodes and visitors is illustrated in Figure 5.3. The visitor is passed to a node using the node’s accept(Visitor) method. The node then invokes a type-specific visit method on the visitor, passing itself as the parameter. The preVisit(ASTNode) and postVisit(ASTNode) methods of the visitor are used for node type-invariant logic. The preVisit method is invoked by the node just before its type-specific visit method is invoked. The postVisit method is invoked by the
node immediately after its type-specific visit method has returned. This behaviour allows the visitor to execute node type-invariant logic both before and after the type-specific logic for each node.

The ASTVisitor class is abstract, meaning it cannot be instantiated directly. Instead, a new visitor is created that extends it, and overrides the relevant visit methods. The abstract class has default implementations for its methods so only those methods needed for the extraction are overridden.

Abstract syntax trees are hierarchical structures, and the visitor traverses them as such. To extract relationships between elements at differing levels, a stack data structure is employed for temporary storage. For example, when visiting a type declaration, a Type node is pushed onto a declaration stack. Whenever a field declaration is encountered in the body of the type declaration, the Type node is retrieved from the stack to create an edge between it and the Field node. Similarly, when visiting an expression, an Expression node is pushed onto an expression stack. Unlike declarations, expressions are handled in a bottom-up manner. An expression is either contained in another expression or a statement. When a visit to an expression is finished, an Expression node must be pushed onto the expression stack. The node is retrieved during a visit to another expression or a statement closer to the tree’s root. Listing 5.2 on the next page gives the visitor’s stack implementation, which uses a linked list. These methods are referenced extensively in the sections that follow.

The next few sections describe the visitor implementation for several key abstract syntax tree nodes. These are divided into three categories: declarations, flows, and comparisons. Declaration nodes involve names for source code elements, such as types, methods, and variables. Flow nodes involve data and control flow source
Listing 5.2 Stack implementation for the visitor’s temporary storage.

```java
private void push(LinkedList<Tuple> context, Tuple tuple) {
    context.addLast(tuple);
}
private Tuple pop(LinkedList<Tuple> context) {
    return context.removeLast();
}
private Tuple peek(LinkedList<Tuple> context) {
    return context.getLast();
}
```

code elements, such as assignments, method invocations, and conditional expressions. Comparison nodes involve source code elements having relational operators.

### 5.2.3 Declarations

A declaration associates an identifier with a source code element within some scope. A declaration’s scope is the textual region within which the declared element can be referred to by its identifier [15].

Java has several kinds of declarations, but only four are relevant to the extraction process: types, methods, fields, and local variables. A type declaration introduces an interface or class into a Java program. A method declaration introduces an invokable operation into a type. Optionally, the method declaration may declare formal parameters that introduce local variables into the method. These variables are implicitly assigned the values of expressions passed as actual parameters during a invocation of the method. Lastly, a field declaration introduces a member variable into a type.
Types

There are two kinds of type declarations in the second edition of the Java Language Specification: classes and interfaces. An interface type may be extended by another interface type, and it may be implemented by a class type. A class type may implement any number of interface types, but it may extend at most one class type. This type hierarchy has an interesting consequence. A reference to an interface type can be used to refer to any of its implementing class types or its extending interface types. Likewise, a reference to a class type can be used to refer to any of its extending class types.

It is not always apparent from the source code which type will be used at run-time. Nevertheless, it is possible to determine from its declaration the types that could refer to a given type. For example:

```java
public class Foo extends Bar implements IFooBar {
    ...
}
```

This is a class type declaration named Foo. Within the declaration are the keywords `extends` and `implements`. The first keyword states that Foo extends the class Bar. The second keyword states that Foo also implements the interface IFooBar. This means any references to a Bar or an IFooBar may also be used to reference a Foo. Note also, that Bar might extend a class or implement an interface, and that IFooBar might extend an interface. If so, references to those extended classes or interfaces may also be used to reference a Foo.

The concern of the extraction is whether a method declared by a type overrides a method declared by one of the types that it extends or implements. This is captured
Listing 5.3 Visiting a type declaration to extract overriding methods.

```java
public boolean visit(TypeDeclaration node) {
    ITypeBinding binding = node.resolveBinding();
    ArrayList<IMethodBinding> methodBindings = new ArrayList<IMethodBinding>();
    ITypeBinding superType = binding.getSuperclass();
    if (superType != null) {
        methodBindings.addAll(Arrays.asList(superType.getDeclaredMethods()));
    }
    ITypeBinding[] interfaces = binding.getInterfaces();
    for (int i = 0; i < interfaces.length; i++) {
        methodBindings.addAll(Arrays.asList(interfaces[i].getDeclaredMethods()));
    }
    IMethodBinding[] declaredMethodBindings = binding.getDeclaredMethods();
    for (IMethodBinding declaredMethodBinding : declaredMethodBindings) {
        for (IMethodBinding methodBinding : methodBindings) {
            if (declaredMethodBindingoverrides(methodBinding)) {
                Tuple declaredMethod =
                    Tuple.singleton(declaredMethodBinding.getJavaElement());
                Tuple method = Tuple.singleton(methodBinding.getJavaElement());
                factbase.insert(Relations.OVERRIDES, declaredMethod, method);
            }
        }
    }
    push(scope, resolveName(node.getName()));
    List<BodyDeclaration> bodyDeclarations = node.bodyDeclarations();
    for (BodyDeclaration bodyDeclaration : bodyDeclarations) {
        bodyDeclaration.accept(this);
    }
    pop(scope);
    return false;
}
```

in the `overrides` relation. If so, a method invocation in the source code may actually
be invoking an overriding implementation at run-time. This capability, known as
polymorphism, is used in object-oriented languages, like Java, to allow a method to
be invoked without the caller knowing the run-time type of the callee.

Listing 5.3 shows a visitor implementation for extracting overriding methods from
a type declaration. Lines 4–7 retrieve methods from the class type that the declared
type extends, if one exists. Lines 8–11 retrieve methods from the interface types that
the this declared type implements. Lines 13–22 determine which of the this type’s methods override any of the methods gathered earlier. When an overriding method is found, a fact is inserted into the factbase for the overrides relation (Line 19). Lines 23–28 resolve the type declaration to a Type node and push it onto the declaration stack for use while visiting declarations within this type declaration’s scope. Once the body declarations have all been visited, the Type node is removed from the stack.

Fields

A field declaration introduces one or more variables into a type declaration. For example:

```java
public class Foo {
    private int x;
    private boolean y, z;
}
```

There are two field declarations in this class. The first declaration introduces a variable named x. The second declaration introduces variables named y and z. Since more than one variable can appear in a declaration, it is divided into a series of fragments.

Listing 5.4 on the next page shows a visitor implementation for extracting fields from a field declaration. Line 2 retrieves the Type node for the type declaring this field from the declaration stack. This node was pushed onto the stack while visiting the type’s declaration. Lines 3–8 retrieve fragments for the variables contained in the field declaration, visiting each fragment to generate a Field node. The Field nodes are removed from the expression stack, and a fact is inserted into the factbase for the isFieldOf relation.
CHAPTER 5. FACT EXTRACTION

Listing 5.4 Visiting a field declaration to extract fields.

```
1 public boolean visit(FieldDeclaration node) {
2     Tuple type = peek(declarations);
3     List<VariableDeclarationFragment> fragments = node.fragments();
4     for (VariableDeclarationFragment fragment : fragments) {
5         fragment.accept(this);
6         Tuple field = pop(expressions);
7         factbase.insert(Relations.IS_FIELD_OF, field, type);
8     }
9     return false;
10 }
```

Methods

A method declaration introduces an invokable operation into a type declaration. This declaration may include a fixed number of formal parameters. For example:

```
public class Foo {
    public boolean isEven(int x) {
        return x % 2 == 0;
    }
}
```

There is one method declaration in this class. It declares an operation named `isEven`, with a formal parameter named `x`. The formal parameter declares a variable whose scope is limited to the method’s body.

Listing 5.5 on the next page shows a visitor implementation for extracting the formal parameter variables from a method declaration. Line 2 creates a Method node for the declaration. Lines 3–9 retrieve the formal parameters of the method to create LocalVariable nodes for the local variables. A FormalParameter node is created for each parameter position. A fact is then inserted into the factbase for the `isVariableOf` relation. Lines 10–15 retrieve the method declaration’s body, if one exists. Recall that interface methods do not have implementations. When a body is present, the
Listing 5.5 Visiting a method declaration to extract formal parameter variables.

```java
public boolean visit(MethodDeclaration node) {
    Tuple method = resolveName(node.getName());
    int position = 0;
    List<SingleVariableDeclaration> parameters = node.parameters();
    for (SingleVariableDeclaration parameter : parameters) {
        Tuple variable = resolveName(parameter.getName());
        Tuple formalParameter = Tuple.pair(method, position++);
        factbase.insert(Relations.IS_VARIABLE_OF, variable, formalParameter);
    }
    Block body = node.getBody();
    if (body != null) {
        push(declarations, method);
        body.accept(this);
        pop(declarations);
    }
    return false;
}
```

Method node is pushed onto the declaration stack for use while visiting statements in the method’s body. Once the body has been visited, the node is removed from the stack.

5.2.4 Flows

A flow involves a data transfer or control expression. Data transfer expressions move a value from one or more variables to another variable. Control expressions are the conditions of control statements that cause a program to branch or loop.

Data Flow

Assignment is the most basic form of data flow. Assignments are expressions having one of the following operators: =, +=, -=, ++, --. The first three are binary operators. The last two are unary, appearing before or after a variable to increment or decrement
its value. These unary operators contain implicit assignments. Besides assignment
expressions, data can also transferred by method invocation. The actual parameters
to an invocation are implicitly assigned to the formal parameters of the method. Also,
if the method returns a value, it can be assigned to a variable.

The following example illustrates several kinds of data flow:

```java
public int foo(int x) {
    return ++x;
}

public void bar() {
    int y = 0;
    y = foo(y);
}
```

The method `bar()` contains a local variable declaration with an initializer, an actual
parameter to a method invocation, and an assignment to the local variable from the
invocation. The method `foo()` contains an increment, whose value is returned by the
method. Together, there are four data flows:

1. 0 to y;
2. y to x;
3. x plus 1 to x;
4. x to y.

Listing 5.6 on the next page shows a visitor implementation for extracting explicit
assignments from an assignment expression. Lines 2–3 retrieve the left operand of the
assignment, which is an expression. The visitor then traverses the operand and pushes
an Expression node for it onto the expression stack. When the visitor returns, it gets
the node from the expression stack without removing it. Thus, if the assignment is
Listing 5.6 Visiting an assignment expression to extract an explicit assignment.

```java
public boolean visit(Assignment node) {
    node.getLeftHandSide().accept(this);
    Tuple leftOperand = peek(expressions);
    node.getRightHandSide().accept(this);
    Tuple rightOperand = pop(expressions);
    Tuple operator = Tuple.singleton(node.getOperator());
    Tuple assignment = Tuple.triple(leftOperand, rightOperand, operator);
    factbase.insert(Relations.IS_LEFT_OPERAND_OF, leftOperand, assignment);
    factbase.insert(Relations.IS_RIGHT_OPERAND_OF, rightOperand, assignment);
    factbase.insert(Relations.IS_OPERATOR_OF, operator, assignment);
    return false;
}
```

Listing 5.7 Visiting a postfix expression to extract an implicit assignment.

```java
public boolean visit(PostfixExpression node) {
    node.getOperand().accept(this);
    Tuple operand = peek(expressions);
    Tuple operator = Tuple.singleton(node.getOperator());
    factbase.insert(Relations.IS_OPERATOR_OF, operator, operand);
    return false;
}
```

if itself an operand of another expression, the left operand of this assignment becomes an operand of the other. Alternatively, the left operand Expression node could be removed from the stack and an Expression node for the assignment pushed onto the stack. Doing so, however, would increase the complexity of data flow analysis. Lines 4–5 perform the same task for the right operand. Lines 6–7 create an Operator node for the operator and an Expression node for the assignment expression. Lines 8–10 insert facts into the factbase for the isLeftOperandOf, isRightOperandOf, and isOperatorOf relations.

Listing 5.7 shows a visitor implementation for extracting an implicit assignment from a postfix expression. Lines 2–3 retrieve the operand of the expression, which is
Listing 5.8 Visiting a method invocation to extract parameters.

```java
public boolean visit(MethodInvocation node) {
    Tuple method = resolveName(node.getName());
    List<Expression> arguments = node.arguments();
    int position = 0;
    for (Expression argument : arguments) {
        argument.accept(this);
        Tuple actualParameter = pop(expressions);
        Tuple formalParameter = Tuple.pair(method, position++);
        factbase.insert(Relations.IS_ACTUAL_PARAMETER_OF, actualParameter,
                        formalParameter);
        factbase.insert(Relations.IS_FORMAL_PARAMETER_OF, formalParameter,
                        method);
    }
    return false;
}
```

itself an expression. The visitor then traverses the operand and pushes an Expression node for it onto the expression stack. When the visitor returns, it gets the node from the expression stack without removing it. Thus, if the postfix expression is itself an operand of another expression, this expression’s operand becomes an operand of the other. Alternatively, the operand node could be removed from the stack and another Expression node containing the operand and operator could be pushed onto the expression stack. Doing so, however, would increase the complexity of data flow analysis. Lines 4–5 create an Operator node for the operator, and insert a fact into the factbase for the isOperatorOf relation. Notice that the isLeftOperandOf and isRightOperandOf relations are not used, since postfix operators are unary.

Listing 5.8 shows a visitor implementation for extracting parameters from a method invocation expression. Line 2 creates a Method node for the invoked method. Line 3 retrieves the arguments of the invocation expression, which are themselves expressions. The visitor then traverses the argument expressions, pushing an Expression node onto the expression stack for each. When the visitor returns, it removes the node
Listing 5.9 Visiting a method return statement to extract a return value.

```java
public boolean visit(ReturnStatement node) {
    Expression expression = node.getExpression();
    if (expression != null) {
        expression.accept(this);
        Tuple returnValue = pop(expressions);
        Tuple method = peek(declarations);
        factbase.insert(Relations.RETURNS, method, returnValue);
    }
    return false;
}
```

from the expression stack. Line 8 creates a FormalParameter node for the parameter position. This node is equivalent to the one extracted from the method’s declaration on Line 7 in Listing 5.5 on page 88. Lines 9–12 insert facts into the factbase for the `isActualParameterOf` and `isFormalParameterOf` relations.

Listing 5.9 shows a visitor implementation for extracting a return value from a return statement. Line 2 retrieves the expression being returned, if one exists. For methods that have a `void` return type, return statements cannot have an expression [15]. Lines 4–5 cause the visitor to visit the expression, which pushes an Expression node onto the expression stack. The node is then removed from the stack. Line 6 retrieves the Method node containing this return statement from the declaration stack. Line 7 inserts a fact into the factbase for the `returns` relation.

Listing 5.9 shows a visitor implementation for extracting a subscript from an array access expression. Line 2–5 retrieve the array and index, which are themselves expressions, and tells the visitor to visit them. In each case, an Expression node is pushed onto the expression stack, which is then removed when the visitor returns. Line 6 creates an Expression node for the array access expression. A fact is then
Listing 5.10 Visiting an array access expression to extract a subscript.

```java
public boolean visit(ArrayAccess node) {
    node.getArray().accept(this);
    Tuple array = pop(expressions);
    node.getIndex().accept(this);
    Tuple index = pop(expressions);
    Tuple access = Tuple.pair(array, index);
    factbase.insert(Relations.IS_INDEX_OF, index, access);
    push(expressions, access);
    return false;
}
```

Listing 5.11 Visiting a conditional expression to extract a condition.

```java
public boolean visit(ConditionalExpression node) {
    node.getExpression().accept(this);
    Tuple condition = pop(expressions);
    node.getThenExpression().accept(this);
    Tuple thenExpression = pop(expressions);
    node.getElseExpression().accept(this);
    Tuple elseExpression = pop(expressions);
    Tuple conditional = Tuple.triple(condition, thenExpression, elseExpression);
    factbase.insert(Relations.IS_CONDITION_OF, condition, conditional);
    factbase.insert(Relations.IS_LEFT_OPERAND_OF, thenExpression, conditional);
    factbase.insert(Relations.IS_RIGHT_OPERAND_OF, elseExpression, conditional);
    push(expressions, conditional);
    return false;
}
```

inserted into the factbase for the `isIndexOf` relation. Line 8 pushes the array access Expression node onto the expression stack to be consumed by an expression or statement closer to the tree’s root.

Control Flow

Listing 5.11 shows a visitor implementation for extracting a condition from a conditional expression. Lines 2–7 retrieve and visit the three parts of a conditional
expression: the condition, the then-expression, and the else-expression. The visitor visits each subexpression using its accept method, pushing an Expression node onto the expression stack, then removing it when the accept method returns. Line 8 then creates an Expression node for the conditional expression using the subexpressions’ nodes. Lines 9–11 insert the facts into the factbase for the isConditionOf, isLeftOperandOf, isRightOperandOf relations.

5.2.5 Comparisons

A comparison is an infix expression having one of the relational operators: $>$, $<$, $\geq$, $\leq$, $\approx$, $\neq$. For example:

```java
if (x > y && y >= 0) {
    ...
}
```

There are three infix expressions in this statement, but only two are comparisons. The first comparison tests whether the value of the variable $x$ is greater than the value of the variable $y$. The second comparison tests whether the value of the variable $y$ is greater than or equal to zero. The other infix expression is a logical conjunction of the two comparisons.

Listing 5.12 on the next page shows a visitor implementation for extracting comparisons from an infix expression. Line 2–5 retrieve the left and right operands of the infix expression, which are themselves expressions. As each operand expression is visited using its accept method, an Expression node is pushed onto the expression stack. This node is then removed when the accept method returns. Lines 6–7 create an Operator node for the infix expression’s operator. The Expression and Operator nodes are combined to form a 3-tuple for the infix expression. Lines 8–10 insert
Listing 5.12 Visiting an infix expression to extract comparisons.

```java
public boolean visit(InfixExpression node) {
    node.getLeftOperand().accept(this);
    Tuple leftOperand = pop(expressions);
    node.getRightOperand().accept(this);
    Tuple rightOperand = pop(expressions);
    Tuple operator = Tuple.singleton(node.getOperator());
    Tuple expression = Tuple.triple(leftOperand, rightOperand, operator);
    factbase.insert(Relations.IS_LEFT_OPERAND_OF, leftOperand, expression);
    factbase.insert(Relations.IS_RIGHT_OPERAND_OF, rightOperand, expression);
    factbase.insert(Relations.IS_OPERATOR_OF, operator, expression);

    List<Expression> extendedOperands = node.extendedOperands();
    for (Expression extendedOperand : extendedOperands) {
        leftOperand = expression;
        extendedOperand.accept(this);
        rightOperand = pop(expressions);
        expression = Tuple.triple(leftOperand, rightOperand, operator);
        factbase.insert(Relations.IS_LEFT_OPERAND_OF, leftOperand, expression);
        factbase.insert(Relations.IS_RIGHT_OPERAND_OF, rightOperand, expression);
        factbase.insert(Relations.IS_OPERATOR_OF, operator, expression);
    }

    push(expressions, expression);
    return false;
}
```
the facts into the factbase for the isLeftOperandOf, isRightOperandOf, and isOperatorOf relations. A list of extended operands reduces the overhead of deeply nested infix expressions of the same operator. Rather than recursively visiting these infix expressions, the flattened operands are visited iteratively in Lines 11–20. An Expression node for the top-level infix expression is pushed onto the expression stack to be consumed by the expression or statement containing this one.

5.3 Summary

In this chapter, the metamodel developed from the empirical study was transformed into a graph schema. The schema describes the source code elements and relationships as sets of nodes and edges. Instances of the schema are graphs defining binary relations over the elements. The fact manipulation and pattern matching processes in the next chapters use these graphs to recover the tuning parameters.

An implementation of extraction was demonstrated using Eclipse’s Java Development Tools. As a platform for Java development, a core functionality of JDT is parsing. The JDT’s parser generates abstract syntax trees. These trees are then traversed using an implementation of the visitor design pattern to extract the relationships specified in the graph schema.
Chapter 6

Fact Manipulation

The facts generated during the extraction process form the strong relationship graph. While it is possible to perform the pattern matching on this graph, the patterns would be more complex. Instead, the facts are first manipulated to form a weak relationship graph. These manipulations abstract away some of the detail and help to overcome the indirection induced by the object-oriented paradigm. This is particularly important for approximating the dataflows of methods in a type hierarchy.

This chapter discusses the manipulations in terms of two types of indirection: method invocation, and polymorphism. In object-oriented languages such as Java, behaviours are defined as discrete blocks of executable code called methods. A method invocation means leaving one block to execute another, and then returning. Data can be transferred into a method using parameters, and transferred out of a method using return values. Polymorphism takes this a step further. Object data types are defined hierarchically, allowing many implementations for an interface, which can be extended further still to specialize the behaviour. Methods in a type hierarchy can be defined with the same name and parameter types. Depending on the runtime type of
an object, a different method may be invoked than the one defined in the variable’s declared type.

As mentioned in Chapter 5, the facts are maintained as a set of binary relations. These relations are manipulated using a relational programming language, RML [5]. RML is the programming language of CrocoPat, a tool for the relational analysis of software. The language is based on first-order predicate calculus.

In the sections that follow, the manipulations are presented as relational expressions in RML. The result of the expression is illustrated as a graph containing a new (dashed) edge corresponding to the manipulated relation. As well, source code examples are provided to demonstrate what the particular manipulation is meant to handle.

### 6.1 Method Invocations

During a method invocation, data is transferred into the method using parameters. For example:

```java
public void f(int y) {
  ...
}

public void g() {
  f(x + 1);
}
```

The method `f` has one formal parameter, `y`. The method `g` invokes the method `f`, passing the expression `x + 1` as an actual parameter. The extraction would generate an `isVariableOf` fact for the formal parameter, and an `isActualParameterOf` fact for the expression. Implicitly, `y` is assigned the value of `x + 1` when `f` is invoked. To
Figure 6.1: Relating variables of formal parameters to actual parameters.

make this explicit, an \textit{isAssignedValueOf} relation can be created using the relational expression:

\[
isAssignedValueOf(v,e) := \text{EX}(f, \text{isVariableOf}(v,f) \land \text{isActualParameterOf}(e,f));
\]

The relational expression checks for bindings for the terms \(v\) and \(e\) that satisfy the existential quantifier. In this way, the variables of formal parameters that have actual parameters are related directly to those actual parameters. Figure 6.1 depicts how the relational expression changes the graph. An edge is created between a LocalVariable node \(v\) and an Expression node \(e\) if they are connected by a FormalParameter node \(f\) through the edges shown.

During a method invocation, data is transferred out of the method by return values. For example:

```java
public int f() {
    return x;
}

public int g() {
    return f();
}
```

The method \(f\) returns the value of a variable \(x\). The method \(g\) returns the value returned by the method \(f\). The extraction would generate a \textit{returns} fact for \(f\) and \(x\),
and one for \( g \) and \( f \). However, because \( g \) returns the value returned by \( f \), \( g \) indirectly returns \( x \). To make this explicit, the \textit{returns} relation can be redefined using the relational expression:

\[
\text{returns}(m_1, e) := \text{returns}(m_1, e) \mid \text{EX}(m_2, \text{returns}(m_1, m_2) \& \text{returns}(m_2, e));
\]

The relational expression checks for bindings for the terms \( m_1 \) and \( e \) that satisfy either the first relational expression, or the second. In this way, expressions that are returned directly, or indirectly, by a method are related to that method. Figure 6.2 depicts how the relational expression changes the graph. An edge is created between a Method node \( m_1 \) and an Expression node \( e \) if they are connected by a Method node \( m_2 \) through the edges shown.

The \textit{returns} relation may need to be generalized to any level of indirection. This can be accomplished using a transitive closure:

\[
\text{returns}(m, e) := \text{TC}(\text{returns}(m, e));
\]

Method return values must be explicitly assigned to variables, or used in expressions, otherwise they are ignored. For example:
The method \( f \) returns the value of the variable \( x \). The method \( g \) compares the return value of \( f \) to 0, and assigns the return value to the variable \( y \) if it is greater than 0. Here, \( f \) is invoked as the left operand of an infix expression, and as the right operand of an assignment. The extraction would generate a \( \text{return} \) fact for \( f \) and \( x \), an \( \text{isLeftOperandOf} \) fact for \( f \) and the infix expression, and an \( \text{isRightOperandOf} \) fact for \( f \) and the assignment. Since \( f \) returns \( x \), \( x \) is indirectly an operand of both expressions as well. To make this explicit, the \( \text{isLeftOperandOf} \) and \( \text{isRightOperandOf} \) relations can be redefined using the relational expressions:

\[
\text{isLeftOperandOf}(e_1, e_2) := \\
\hspace{1cm} \text{isLeftOperandOf}(e_1, e_2) \mid \text{EX}(m, \text{isLeftOperandOf}(m, e_2) \& \text{return}(m, e_1)); \\
\text{isRightOperandOf}(e_1, e_2) := \\
\hspace{1cm} \text{isRightOperandOf}(e_1, e_2) \mid \text{EX}(m, \text{isRightOperandOf}(m, e_2) \& \text{return}(m, e_1));
\]

The relational expressions check for bindings for the terms \( e_1 \) and \( e_2 \) that satisfy either the first relational expression, or the second. Expressions returned by methods are now directly related to expressions in which those methods are operands. Figure 6.3 on the next page depicts how the relational expressions change the graph. An edge is created between an Expression node \( e_1 \) and an Expression node \( e_2 \) if they are connected by a Method node \( m \) through the edges shown.

Methods having boolean return values may be used directly as conditions. For example:
Figure 6.3: Relating method return values to expressions in which the method is an operand.

```java
public boolean f() {
    return x;
}

public void g() {
    if (f()) {
        ...
    }
}
```

The method `f` returns a boolean variable `x`. The method `g` invokes `f` as the condition of an `if` statement. The extraction would generate a `returns` fact for `f` and `x`, and an `isConditionOf` fact for `f`. Since `f` returns `x`, `x` is indirectly a condition of the `if` statement. To make this explicit, the `isConditionOf` relation can be redefined using the relational expression:

```latex
isConditionOf(e1,e2) :=
    isConditionOf(e1,e2) |
    EX(m, isConditionOf(m,e2) & returns(m,e1));
```

The relational expression checks for bindings for the terms `e1` and `e2`. Expressions returned by methods are now directly related to expressions for which those methods
are conditions. Figure 6.4 depicts how the relational expression changes the graph. An edge is created between an Expression node $e1$ and an Expression node $e2$ if they are connected by a Method node $m$ through the edges shown.

6.2 Polymorphism

The extraction process generates \textit{overrides} facts for methods that override one another. However, it only does this for methods that directly override one another. Should a method override a method that overrides a method, no fact is generated for the first and last methods. As it turns out, the \textit{overrides} relation is transitive. If a method $t$ overrides a method $u$, and $u$ overrides a method $v$, it is also true that $t$ overrides $v$.

The \textit{overrides} facts can be manipulated to generate a new relation that also captures those methods that are indirectly overridden:

\[
\text{overrides}(t,v) := \text{overrides}(t,v) \mid \text{EX}(u, \text{overrides}(t,u) \& \text{overrides}(u,v));
\]
This relational expression redefines the \textit{overrides} relation, such that: a method $t$ overrides a method $v$ if $t$ overrides $v$, or there exists a method $u$ that $t$ overrides, and that overrides $v$. As depicted in Figure 6.5, the result is a new \textit{overrides} edge in the graph. A transitive closure can be used to extend this to any level of indirection:

\[
\text{overrides}(t,v) := \text{TC}(\text{overrides}(t,v));
\]

These manipulations are important for approximating the polymorphic behaviour of objects. Consider the following example:

```java
public interface I {
    public int f();
}

public class A implements I {
    public int f() {
        return 0;
    }
}

public class B extends A {
    private int x;

    public int f() {
        return x;
    }
}
The interface \( I \) is implemented by the class \( A \). The class \( A \) is extended by the class \( B \). At runtime, a variable of type \( I \) is allowed to reference an object of type \( A \) or \( B \). Thus, invoking \( f() \) on that object potentially executes the behaviour from either type. These relationships are captured by the \textit{overrides} facts. For this example, an \textit{overrides} fact would be generated for the methods of \( A \) and \( I \), and for the methods of \( B \) and \( A \). However, the methods of \( B \) and \( I \) are not directly related. Manipulating the facts as described above captures the relationship explicitly.

The \textit{overrides} facts alone are not very useful. But, when combined with other facts, it is possible to approximate the dataflows of overridden methods in a type hierarchy. The data flow out of the methods through return values, and into them through parameters.

The return values of methods are captured by \textit{returns} facts. These facts can be combined with the manipulated \textit{overrides} facts to generate a new relation that also captures the expressions that are returned by invocations of overridden methods:

\[
\text{returns}(t,v) := \text{returns}(t,v) \mid \text{EX}(u, \text{returns}(u,v) \land \text{overrides}(u,t));
\]

This relational expression redefines the \textit{returns} relation, such that: a method \( t \) returns an expression \( v \) if \( t \) returns \( v \), or there exists a method \( u \) that overrides \( t \), and that returns \( v \). As depicted in Figure 6.6 on the next page, the result is a new \textit{returns} edge in the graph. Consider a class that invokes the method \( f() \) from the previous example:

```java
public class C {
    private I i;
}```
Depending on the runtime value of $i, f()$ could return 0, or the value of the field $x$ from an instance of $B$. By manipulating the facts as described above, both cases are captured explicitly.

The parameters of methods are captured by the $isActualParameterOf$, $isFormalParameterOf$, and $isPositionOf$ facts. These facts can be combined with the manipulated $overrides$ facts to generate a new relation for expressions that are, or could be, passed as parameters to methods:

```java
public void g() {
    if (i.f() > 0) {
        ...
    }
}
```

```
isActualParameterOf(t,v) :=
isActualParameterOf(t,v) |
EX(u,m,n,i,
isActualParameterOf(t,u) &
isFormalParameterOf(u,m) &
overrides(n,m) &
isFormalParameterOf(v,n) &
isPositionOf(i,u) &
isPositionOf(i,v));
```
This relational expression redefines the \textit{isActualParameterOf} relation, such that: an expression \( t \) is an actual parameter of a formal parameter \( v \) if \( t \) is an actual parameter of \( v \), or there exists a formal parameter \( u \), to which \( t \) is an actual parameter, at position \( i \) of a method \( m \) overridden by a method \( n \) having a formal parameter \( v \) at the same position as \( u \). As depicted in Figure 6.6 on the previous page, the result is a new \textit{isActualParameterOf} edge in the graph.

```java
public interface I {
    public void f(int y);
}

public class A implements I {
    private int x;

    public void f(int y) {
        x = y;
    }
}

public class B {
    I i;

    public void g() {
        i.f(0);
    }
}
```
The manipulations thus far have dealt with approximating where data are transferred. But, equally important is how data are transferred. The operators used in data transfer expressions are a key aspect of the tuning parameter patterns. The extraction handles expressions that contain operators in a consistent manner—if the expression is unary, the operator and expression are related directly; if the expression is binary, the operator and operands are related indirectly through the expression in which they appear. The operator manipulations described in the next few sections help to relate operands directly, as well as to normalize certain expressions into a canonical form.

### 6.3 Assignments

Assignments are expressions having one of the assignment operators. For example:

```plaintext
y = z;
\textcolor{red}{x} += y;
\textcolor{red}{z} -= 1;
```

The first statement contains a simple assignment of the variable `z` to the variable `y`. The second statement contains an assignment of the variable `y` to the variable `x` using a compound assignment operator. Semantically, the value of `x` is being incremented by the value of `y`. The third statement contains an assignment of the numeric constant 1 to the variable `z` using a compound assignment operator. Semantically, the value of `z` is being decremented by 1. For each assignment, the extraction generates three facts: an `isLeftOperandOf` fact for the variable to the operator’s left, an `isRightOperandOf`
fact for the expression to the operator’s right, and an isOperatorOf fact for the operator. The operands of the assignments are related indirectly by these facts. The relational expressions relate them directly, while also making their semantics more explicit:

\[
\text{isAssignedValueOf}(e_1,e_2) := \\
\quad \text{EX}(e_3, \\
\quad \quad \text{isLeftOperandOf}(e_1,e_3) \land \\
\quad \quad \text{isRightOperandOf}(e_2,e_3) \land \\
\quad \quad \text{isOperatorOf}("=",e_3)); \\
\text{increments}(e_1,e_2) := \\
\quad \text{EX}(e_3, \\
\quad \quad \text{isLeftOperandOf}(e_2,e_3) \land \\
\quad \quad \text{isRightOperandOf}(e_1,e_3) \land \\
\quad \quad \text{isOperatorOf}("+=",e_3)); \\
\text{decrements}(e_1,e_2) := \\
\quad \text{EX}(e_3, \\
\quad \quad \text{isLeftOperandOf}(e_2,e_3) \land \\
\quad \quad \text{isRightOperandOf}(e_1,e_3) \land \\
\quad \quad \text{isOperatorOf}("-=",,e_3));
\]

The first relational expression defines a relation isAssignedValueOf. It checks for bindings for the terms $e_1$ and $e_2$, such that there exists an $e_3$ that has $e_1$ as its left operand, $e_2$ as its right operand, and $=$ as its operator. The second relational expression defines a relation increments. It checks for bindings for the terms $e_1$ and $e_2$, such that there exists an $e_3$ that has $e_2$ as its left operand, $e_1$ as its right operand, and $+=\text{ as its operator. The third relational expression defines a relation decrements. It checks for a bindings for the terms $e_1$ and $e_2$, such that there exists an $e_3$ that has $e_2$ as its left operand, $e_1$ as its right operand, and $-=$ as its operator.}

Figure 6.8 on the next page depicts how the relational expressions manipulate the graph. An edge is created between an Expression node $e_1$ and an Expression node $e_2$, if they are connected by an Expression node $e_3$ through the edges shown and $e_3$ is connected to an Operator node $o$ containing the operator $=\text{, }+=\text{, or }-=\text{.}
In addition to the aforementioned explicit assignments, there are also implicit ones. For example:

```plaintext
x++; y--; 
```

The variable `x` is incremented in the first statement using a postfix operator. The variable `y` is decremented in the second statement using a prefix operator. The extraction generates an `isOperatorOf` fact for each, relating the operator to the expression being incremented or decremented. Notice that an equivalent pair of statements would be:

```plaintext
x += 1; y -= 1; 
```

There are subtle differences between the evaluation semantics of each set of statements, but these are not important for the pattern matching process. The assignments are thus captured explicitly, using the relational expressions:

```plaintext
increments("1", e) := isOperatorOf("++", e); decrements("1", e) := isOperatorOf("--", e); 
```
These relational expressions resemble those used above for the compound assignments, except that instead of checking for bindings for two terms, the first term is a string literal. The expressions check for bindings for $e$, such that $e$ has $++$ or $--$ as its operator.

Figure 6.9 depicts how the relational expressions manipulate the graph. An edge is created between an Expression node containing the numeric constant “1” and an Expression node $e$, if $e$ is connected to an Operator node $o$ that contains the operator $++$ or $--$.

6.4 Comparisons

Comparisons are infix expressions having relational operators. They often appear as conditions in if statements. For example:

```java
if (x > y) {
    ...
} else if (x < y) {
    ...
} else {
    ...
}
```

The variables $x$ and $y$ are involved in two comparisons. For the first comparison, the extraction generates an isLeftOperandOf fact for $x$, an isRightOperandOf fact for $y$, and an isOperatorOf fact for the $>$ operator. For the second comparison, the
extraction generates similar facts, except the operands are reversed and the operator is <. The variables x and y are related indirectly by these facts. The relational expressions relate them directly, while also making their semantics more explicit:

\[
\begin{align*}
isComparedGreaterThanOrEqualTo(e_1,e_2) := \\
\quad \text{EX}(e_3, \\
\quad \quad \text{isLeftOperandOf}(e_1,e_3) \& \\
\quad \quad \text{isRightOperandOf}(e_2,e_3) \& \\
\quad \quad (\text{isOperatorOf}("\geq",e_3) \mid \text{isOperatorOf}("\gt",e_3)); \\
isComparedLessThanOrEqualTo(e_1,e_2) := \\
\quad \text{EX}(e_3, \\
\quad \quad \text{isLeftOperandOf}(e_1,e_3) \& \\
\quad \quad \text{isRightOperandOf}(e_2,e_3) \& \\
\quad \quad (\text{isOperatorOf}("\lt",e_3) \mid \text{isOperatorOf}("\leq",e_3)); \\
isComparedGreaterThanOrEqualTo(e_1,e_2) := \\
\quad \text{isComparedGreaterThanOrEqualTo}(e_1,e_2) \mid \\
\quad \text{isComparedLessThanOrEqualTo}(e_2,e_1); \\
isComparedLessThanOrEqualTo(e_1,e_2) := \\
\quad \text{isComparedLessThanOrEqualTo}(e_1,e_2) \mid \\
\quad \text{isComparedGreaterThanOrEqualTo}(e_2,e_1);
\end{align*}
\]

The first relational expression defines a relation \textit{isComparedGreaterThanOrEqualTo}. It checks for bindings for the terms \(e_1\) and \(e_2\), such that there exists an \(e_3\) that has \(e_1\) as its left operand, \(e_2\) as its right operand, and \(\geq\) as its operator. The second relational expression defines a relation \textit{isComparedLessThanOrEqualTo}. It checks for bindings for the terms \(e_1\) and \(e_2\), such that there exists an \(e_3\) that has \(e_1\) as its left operand, \(e_2\) as its right operand, and \(\leq\) as its operator. Semantically, the operators of the first and second relational expressions are inverses of one another. The final two relational expressions capture this by redefining each relation to also include the inverse of the other.

Figure 6.10 on the next page depicts how the relational expressions manipulate the graph. An edge is created between an Expression node \(e_1\) and an Expression node \(e_2\) if they are connected by an Expression node \(e_3\) through the edges shown.
6.5 Conditions

Conditions may be simple boolean expressions, or infix expressions having logical operators. For example:

```java
if (x && (y || z)) {
    ...
}
```

Here, the condition of the `if` statement is composed of two infix expressions: a logical conjunction using the `&&` operator, which is composed of a simple boolean expression and a logical disjunction using the `||` operator. The extraction first generates facts for the disjunction. An `isLeftOperandOf` fact is generated for the variable `y`, an `isRightOperandOf` fact for the variable `z`, and an `isOperatorOf` fact for the `||` operator. It then generates facts for the conjunction. An `isLeftOperandOf` fact for the variable `x`, an `isRightOperandOf` fact for the disjunction, and an `isOperatorOf` fact for the `&&` operator. Finally, it generates an `isConditionOf` fact for the entire expression. Semantically, the truth value of the condition depends on the truth values of its logical subexpressions. This dependency is made explicit by also making the operands of the logical infix expressions be conditions using the relational expressions:
```plaintext
tmp(e1,e2) := FALSE(e1,e2);
WHILE (tmp(e1,e2) != isConditionOf(e1,e2)) {
    tmp(e1,e2) := isConditionOf(e1,e2);
    isConditionOf(e1,e2) :=
    isConditionOf(e1,e2) |
    EX(e3,
        isConditionOf(e3,e2) &
        (isLeftOperandOf(e1,e3) | isRightOperandOf(e1,e3)) &
        (isOperatorOf("&&",e3) | isOperatorOf("||",e3)));
}
```

The relational expressions repeatedly redefine the `isConditionOf` relation until it reaches a fixed point. This is necessary since an operand of a logical infix expression can itself be a logical infix expression. Everytime an infix expression becomes a fact in the relation, its operands become subject to the terms of the relational expression. The relation `tmp` is used to determine when the fixed point is reached. It contains the facts from the previous version of the `isConditionOf` relation.

The core relational expression checks for bindings for the terms `e1` and `e2`, such that `e1` is a condition of `e2`, or there exists an `e3` that is a condition of `e2`, and that has `e1` as its left or right operand and `&&` or `||` as its operator.

Figure 6.11 on the next page depicts how the relational expressions manipulate the graph. An edge is created between an Expression node `e1` and an Expression node `e2` if they are connected by an Expression node `e3` through the edges shown.

### 6.6 Summary

The facts generated during the extraction process can be manipulated using relational programming. While not strictly necessary, these manipulations reduce the complexity of expressing the tuning parameter patterns. The extracted facts form a strong relationship graph that is weakened by the manipulations. Weakening, in this
context, refers to presence of edges in the graph that represent relationships that were not taken directly from the source code. For example, the polymorphic behaviour of methods may be overapproximated in the manipulated facts since it is difficult to determine exactly which behaviour will be invoked at runtime.

The fact manipulations are expressed using RML, a relational programming language developed for the CrocoPat software analysis tool. The language is based on first-order predicate calculus, making it possible to use existential quantification and predicates. The manipulations redefine the existing relations, and define new relations, to create new edges in the relationship graph. These new edges are used by the pattern matching process in the next chapter to identify potential tuning parameters.
Chapter 7

Pattern Matching

In this chapter several of the tuning parameter patterns from Chapter 4 are expressed as RML programs. Whereas the RML programs in Chapter 6 generated new edges in the relationship graph, the objective in this chapter is to query the graph for nodes that are potential tuning parameters. The patterns shown are taken from each of the primary subclassifications of the taxonomy. Together, they demonstrate how the pattern matching process is performed, and the features of RML that support it.

7.1 Patterns

The patterns are expressed using RML, the language used in the previous chapter to express the fact manipulations. The programs query the weak relationship graphs for subgraphs satisfying the tuning parameter patterns. The query result is a set of nodes containing fields that are potential tuning parameters.
Listing 7.1 Querying the graph for instances of the toggle pattern.

toggle(e1) :=
    isFieldOf(e1,_) &
    isConditionOf(e1,_) &
    EX(e2,
        isAssignedValueOf(e1,e2) &
        (returns(e2,_) | isVariableOf(e2,_)));

7.1.1 Toggle

Toggles control behaviour by enabling or disabling it. They are manifested as fields that are evaluated in the condition of an if statement or ?: (conditional) operator.

The RML program shown in Listing 7.1 queries the graph for instances of the toggle pattern. The toggle pattern is defined as an expression $e1$ that: is a field, is a condition of some other expression, and for which there exists an assignment from an expression $e2$ that is either a method invocation, or a formal parameter of a method. This last constraint discriminates against those fields whose state is intrinsic to the class.

Figure 7.1 on the next page depicts a toggle pattern in the weak relationship graph of the toggle stereotype from Chapter 4. The field disableUploadTimeout is a toggle since it is a condition for whether a socket timeout is set, and it has a mutator method that changes its value.

7.1.2 Counter

Counters measure occurrences of events, or actions, using discrete, constant amounts. They are manifested as fields that are incremented, decremented, or both.

The RML program shown in Listing 7.2 on page 120 queries the graph for instances
Figure 7.1: A weak relationship graph containing a toggle (some edges have been omitted for brevity).
of the counter pattern. It first redefines the increments relation to also include those expressions that are indirectly incremented through method invocations, such as:

```java
public int get() {
    return x;
}

public void set(int x) {
    this.x = x;
}

public void increment() {
    set(get() + 1);
}
```

The `increment()` method gets the value of some field `x`, and sets the value of `x` to one more than that value. Decrements are computed in the same manner, but are not shown here. Next, the `isIndexOf` relation is redefined to also include those expressions that are assigned to an array index. For example:

```java
x = y;
z = a[x];
```

The variable `x`, which is assigned the value of some variable `y`, is an index of some array `a`. The variable `y` is therefore also considered an array index. A regular expression for numeric constants is then defined. Finally, the counter pattern is defined as an expression `e1` that: is a field, is not an array index, for which there exists a numeric constant that either increments or decrements it, and for which there does not exist an assignment from a non-constant or formal parameter.

Figure 7.2 on the next page depicts a counter pattern in the weak relationship graph of the counter stereotype from Chapter 4. The field `nCacheMiss` is a counter since it is only ever assigned numeric constants (i.e., 0), and is incremented by a numeric constant (i.e., 1).
Listing 7.2 Querying the graph for instances of the counter pattern.

```
increments(e1, e2) :=
    increments(e1, e2) | EX(e3, e4,
        isAssignedValueOf(e2, e3) &
        isAssignedValueOf(e3, e4) &
        isVariableOf(e3, _) &
        isRightOperandOf(e1, e4) &
        isLeftOperandOf(e2, e4) &
        isOperatorOf("(+)", e4));

isIndexOf(e1, e2) :=
    isIndexOf(e1, e2) | EX(e3,
        isAssignedValueOf(e3, e1) &
        isIndexOf(e3, e2));

numericConstant(e) := "^-?[0-9]+$"(e);

counter(e1) :=
    isFieldOf(e1, _) &
    !isIndexOf(e1, _) &
    EX(e2,
        numericConstant(e2) &
        (increments(e2, e1) | decrements(e2, e1)) &
        !EX(e2,
            isAssignedValueOf(e1, e2) &
            !numericConstant(e2) &
            !isVariableOf(e2, _));
```

Figure 7.2: A weak relationship graph containing a counter (some edges have been omitted for brevity).
7.1.3 Maximum

Maximums maintain the statistical maximum of numerical, or temporal, values over time. They are manifested as fields that are only assigned expressions to which they have been compared as being greater than.

The RML program shown in Listing 7.3 on the next page queries the graphs for instances of the maximum pattern. It first redefines the `isAssignedValueOf` relation such that assignments from numeric constants are ignored. This ensures expressions such as the following are not trivially matched:

```java
if (x < 0) {
    x = 0;
}
```

If `x` were a field, and this were its only assignment, it would trivially be a statistical maximum. In other words, the value for which the statistical maximum is maintained should be non-constant. Next, the increments and decrements are combined with the regular assignments to make the assignment criterion of the pattern as strict as possible. As well, the transitive assignments through formal parameters of methods are included. Lastly, the maximum pattern is defined as an expression `e1` that is a field, and for which there exists an assignment from an expression `e2` that is compared to `e1` as being greater than or equal to. In addition, all such assignments to `e1` must be from expressions that are either compared to it, or from method invocations or formal parameters.

Figure 7.3 on page 123 depicts a maximum pattern in the weak relationship graph of the maximum stereotype from Chapter 4. The field `sessionMaxAliveTime` is a maximum since it is only ever directly, or indirectly, assigned the variable `timeAlive` to which it is compared as being greater than or equal to.
Listing 7.3 Querying the graph for instances of the maximum pattern.

\[
\text{numericConstant}(e) := @"^-?[0-9]+$"(e);
\]
\[
\text{isAssignedValueOf}(e1,e2) :=
\quad \text{isAssignedValueOf}(e1,e2) \&
\quad \neg \text{numericConstant}(e2);
\]
\[
\text{isAssignedValueOf}(e1,e2) :=
\quad \text{isAssignedValueOf}(e1,e2) \mid
\quad \text{increments}(e2,e1) \mid
\quad \text{decrements}(e2,e1);
\]
\[
\text{isAssignedValueOf}(e1,e2) :=
\quad \text{isAssignedValueOf}(e1,e2) \mid
\quad \text{EX}(e3, \quad
\quad \text{isAssignedValueOf}(e1,e3) \&
\quad \text{isAssignedValueOf}(e3,e2) \&
\quad \text{isVariableOf}(e3,_));
\]
\[
\text{maximum}(e1) :=
\quad \text{isFieldOf}(e1,_) \&
\quad \text{EX}(e2, \quad
\quad \text{isAssignedValueOf}(e1,e2) \&
\quad \text{isComparedGreaterThanOrEqualTo}(e2,e1)) \&
\quad \text{FA}(e2, \quad
\quad \text{isAssignedValueOf}(e1,e2) ->
\quad \text{(isComparedGreaterThanOrEqualTo}(e2,e1) \mid
\quad \text{returns}(e2,_) \mid
\quad \text{isVariableOf}(e2,_))));
\]
Figure 7.3: A weak relationship graph containing a maximum (some edges have been omitted for brevity).
Listing 7.4 Querying the graph for instances of the delay pattern.

```plaintext
numericConstant(e) := @"^-?[0-9]+$"(e);

isAssignedValueOf(e1,e2) :=
    isAssignedValueOf(e1,e2) |
    EX(e3,e4,
        isAssignedValueOf(e1,e3) &
        isLeftOperandOf(e2,e3) &
        isRightOperandOf(e4,e3) &
        numericConstant(e4));

isActualParameterOf(e1,p) :=
    isActualParameterOf(e1,p) |
    EX(e2,
        isActualParameterOf(e2,p) &
        isAssignedValueOf(e2,e1));

delay(f) :=
    isFieldOf(f,_) &
    EX(p,
        isActualParameterOf(f,p) &
        isFormalParameterOf(p,"java.lang.Thread.sleep(long)");
```

7.1.4 Delay

Delays prevent behaviours from occurring for some specified length of time. This is usually accomplished by causing the current thread of execution to leave the runnable state. In Java, threads are forced from the runnable state by the `sleep()` method of the `java.lang.Thread` class.

The RML program shown in Listing 7.4 queries the graph for instances of the delay pattern. It first defines a regular expression for numeric constants. It then redefines the `isAssignedValueOf` relation such that assignments from algebraic expressions involving numeric constants have those numeric constants ignored. For example:

```
t = s * 1000;
```

The assignment is treated as though the variable `s` were directly assigned to the
Figure 7.4: A weak relationship graph containing a delay (some edges have been omitted for brevity).

variable $t$. Assignments like these are common when converting a value in seconds to a value in milliseconds, or vice-versa. Next, the $isActualParameterOf$ relation is redefined such that assignments to actual parameters are also considered actual parameters. For example:

\[
\begin{align*}
    t &= s; \\
    \text{Thread.sleep}(t);
\end{align*}
\]

The variable $s$ is treated as though it were an actual parameter of the $\text{sleep()}$ method, in addition to the variable $t$. Lastly, the delay pattern is defined as a field $f$ for which there exists a formal parameter $p$ of the $\text{sleep()}$ method to which $f$ is an actual parameter.

Figure 7.4 depicts a delay pattern in the weak relationship graph of the delay stereotype from Chapter 4. The field $\text{backgroundSleepInterval}$ is a delay since it is an actual parameter of the $\text{sleep()}$ method of the $\text{java.lang.Thread}$ class.
7.2 Summary

The weak relationship graphs generated by fact manipulation in the previous chapter were queried for instances of tuning parameter patterns in this chapter. Patterns for four of the tuning parameter classifications were presented: toggles, counters, maximums, and delays. The patterns for these classifications are considered representative of the others, in that they demonstrate the various characteristics of tuning parameter usage needed to express them.
Chapter 8

Evaluation

The process outlined in the previous chapters for recovering software tuning parameters is evaluated in this chapter. The evaluation examines both the practicality of the approach, and its effectiveness at recovering the tuning parameters from the studied applications, as well as others that were not previously studied.

Recall that the process of automating the recovery is an iterative one. The empirical study leads to a metamodel of the source code, and to patterns of tuning parameter use. The metamodel and patterns are based on observations from the study, which are likely insufficient after a single iteration. For example, there may be source code element relationships that were initially deemed irrelevant, but that become relevant when trying to reduce noise. Consider the counter pattern. The initial pattern is specified as a field being incremented. However, there are numerous cases where array indices match this pattern. Preventing these matches means enhancing the metamodel to represent the array index relationships. Using this new relationship type, the pattern can be refined to ignore them.

Refinements to the patterns emphasize a reduction in the number of false-negatives,
while possibly permitting more false-positives. False-negatives are tuning parameters that are documented, but not recovered. False-positives are tuning parameters that are recovered, but not documented. False-positives can occur either because the recovered parameters are not suitable for tuning, or because the parameters were just not documented.

The evaluation is based on an implementation of the metamodel and patterns developed from the four applications studied in Chapter 4. This same implementation is applied to three other applications. The documented parameters for these applications are only used to validate the results, and not to refine the metamodel or patterns. The remainder of this chapter describes the experimental setup of the evaluation, and the results of the recovery.

8.1 Experimental Setup

The effectiveness of the approach is measured in terms of its accuracy. The accuracy is computed using the traditional information retrieval [26] definitions of recall and precision:

\[
Recall = \frac{|\{Relevant\} \cap \{Retrieved\}|}{|\{Relevant\}|}
\]

\[
Precision = \frac{|\{Relevant\} \cap \{Retrieved\}|}{|\{Retrieved\}|}
\]

The set of relevant tuning parameters contains those that are gathered from an application’s documentation. The set of retrieved tuning parameters contains those that are recovered from an application.
**CHAPTER 8. EVALUATION**

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Domain</th>
<th>Source</th>
<th>License</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jigsaw</td>
<td>2.2.6</td>
<td>Web/Servlet</td>
<td>W3C</td>
<td></td>
<td>101,149</td>
</tr>
<tr>
<td>Hypersonic 2</td>
<td>1.0.57</td>
<td>Database</td>
<td>MPL 1.1</td>
<td></td>
<td>63,283</td>
</tr>
<tr>
<td>Babylon Chat</td>
<td>2.1</td>
<td>Instant Messaging</td>
<td>GPL 2.0</td>
<td></td>
<td>10,292</td>
</tr>
</tbody>
</table>

Table 8.1: Overview of the additional applications.

### 8.1.1 Applications

Besides the four applications from the empirical study, the three applications in Table 8.1 are also included in the evaluation. These applications extend the previous application domains of database management and web serving, as well as introduce a third application domain, instant messaging.

As with the applications from the empirical study, the tuning parameters are gathered from the documentation for the applications. The parameters are then mapped to field declarations, and classified according to the taxonomy. The mappings are listed in Appendix B. Neither of the applications supports JMX, so the documented parameters are primarily those meant for configuration. It is worth noting that the instant messaging application has no documented tuning parameters, but is included because it was a subject of the initial STAC prototype in [9]. The applications are described briefly in the following sections.

**Jigsaw**

Jigsaw is a web server developed by the World Wide Web Consortium (W3C). The W3C’s mandate is to develop technologies, including specifications and software, for the web. It is for this reason that Jigsaw serves as a reference implementation for the HTTP 1.1 standard. As an experimental platform, it is designed to be extensible, yet fully standards-compliant. It was first released to the public in 1996, and is still
under active development. It is written entirely in Java, and is licensed by the W3C under an Open Source Initiative approved license.

**Hypersonic 2**

Hypersonic 2 (H2) is a relational database engine. It is written in Java, and can be embedded into an application, or deployed in a client/server network architecture. It was initially released in 2005, and is still actively developed. It is licensed under the Mozilla Public License.

**Babylon Chat**

Babylon Chat is a collaboration application for communicating over networks. Users can communicate using instant messaging, as well as a whiteboard drawing utility. It is written in Java, and is deployed in a client/server network architecture. It was first released in 1997, and is still under active development. It is licensed under the GNU General Public License.

The next section describes how the software tuning parameter recovery process is implemented as a search tool, and how this implementation is used to evaluate the approach against the applications.

### 8.1.2 Search Plug-in

The extraction, manipulation and pattern matching have been implemented as a tool for searching an application’s source code for tuning parameters. The tool is a plug-in for the Eclipse platform. It extends the platform’s existing search functionality to allow an entire application, or a subset of an application’s classes, to be searched
for a user-defined set of tuning parameter types. The user is able to search for
 tuning parameters matching the toggle, counter, maximum, or delay patterns, or
 any combination of these. The search result is a list of potential tuning parameters,
 organized according to type. The user can then navigate to the field declaration of
 the tuning parameter, or search for all references to it. These automated navigations
 make it easier to verify the validity of the result through inspection.

Despite having specified the fact manipulations and pattern matches as RML
 programs in the previous chapters, they are also implemented in Java to maximize the
 plug-in’s portability. CrocoPat uses an operating system-dependent binary executable
 because it is written in C++. While it is possible to execute such a binary from Java,
 the complexity of distributing and configuring the plug-in would be greatly increased.

Nevertheless, the evaluation in this chapter is still based on the RML programs
 being executed by CrocoPat. For the evaluation, the plug-in only generates the
 factbases for the applications. The facts extracted from an application are written to
 a file in Rigi Standard Format (RSF) [28] to be read by CrocoPat. RSF is a human-
 readable, line-oriented format where the facts are output as whitespace separated
 triples. Once the factbases are generated, a set of shell scripts execute the RML
 programs for manipulating the facts, and for matching the patterns. The set of
 relevant tuning parameters for an application are written to a file beforehand, and
 are compared to the retrieved set automatically by these scripts. The recall and
 precision are computed from the sets, and recorded for later reference.

The results of the evaluation are presented in the next section.
CHAPTER 8. EVALUATION

<table>
<thead>
<tr>
<th>Apache Derby</th>
<th>Apache Tomcat</th>
<th>Babylon Chat</th>
<th>Hypersonic 2</th>
<th>Jigsaw</th>
<th>Mort Bay Jetty</th>
<th>Oracle Berkeley DB Java Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>isActualParameterOf</td>
<td>15,008</td>
<td>4,417</td>
<td>587</td>
<td>4,228</td>
<td>4,232</td>
<td>1,040</td>
</tr>
<tr>
<td>isConditionOf</td>
<td>11,579</td>
<td>7,026</td>
<td>303</td>
<td>3,134</td>
<td>5,332</td>
<td>1,822</td>
</tr>
<tr>
<td>isFieldOf</td>
<td>5,299</td>
<td>2,402</td>
<td>168</td>
<td>1,495</td>
<td>1,652</td>
<td>477</td>
</tr>
<tr>
<td>isFormalParameterOf</td>
<td>8,347</td>
<td>2,918</td>
<td>261</td>
<td>2,209</td>
<td>1,874</td>
<td>626</td>
</tr>
<tr>
<td>isIndexOf</td>
<td>2,537</td>
<td>1,652</td>
<td>37</td>
<td>1,104</td>
<td>1,365</td>
<td>217</td>
</tr>
<tr>
<td>isLeftOperandOf</td>
<td>36,178</td>
<td>20,105</td>
<td>1,070</td>
<td>10,868</td>
<td>14,541</td>
<td>4,564</td>
</tr>
<tr>
<td>isOperatorOf</td>
<td>37,417</td>
<td>20,867</td>
<td>1,117</td>
<td>11,089</td>
<td>14,779</td>
<td>4,631</td>
</tr>
<tr>
<td>isPositionOf</td>
<td>8,347</td>
<td>2,918</td>
<td>261</td>
<td>2,209</td>
<td>1,874</td>
<td>626</td>
</tr>
<tr>
<td>isRightOperandOf</td>
<td>36,178</td>
<td>20,105</td>
<td>1,070</td>
<td>10,868</td>
<td>14,541</td>
<td>4,564</td>
</tr>
<tr>
<td>isVariableOf</td>
<td>6,666</td>
<td>2,451</td>
<td>119</td>
<td>1,794</td>
<td>1,328</td>
<td>417</td>
</tr>
<tr>
<td>returns</td>
<td>5,651</td>
<td>2,424</td>
<td>15</td>
<td>1,570</td>
<td>1,554</td>
<td>509</td>
</tr>
<tr>
<td>overrides</td>
<td>5,104</td>
<td>3,439</td>
<td>191</td>
<td>2,390</td>
<td>2,097</td>
<td>763</td>
</tr>
<tr>
<td>TOTAL</td>
<td>178,311</td>
<td>90,724</td>
<td>5,199</td>
<td>52,958</td>
<td>65,169</td>
<td>20,256</td>
</tr>
<tr>
<td>TIME (sec)</td>
<td>92</td>
<td>35</td>
<td>1</td>
<td>11</td>
<td>24</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 8.2: Total number of facts extracted from each application. Time is based on Eclipse 3.3.1 running on an Intel 2 GHz processor with a 512 MB heap size, and is computed using `System.currentTimeMillis()`.

8.2 Results

The first phase is the fact extraction using the search plug-in. Projects are created in an Eclipse workspace for each of the applications. A project is a logical container for the application’s source code. The plug-in is then invoked on the project to extract the facts from the source code. The number and kinds of facts extracted from the applications are listed in Table 8.2.

The total number of facts for the applications is always less than the number of lines of code. However, it is not proportional to the number of lines of code. For example, the ratio of lines of code to facts for Jigsaw is nearly 2:1, while the ratio for H2 is nearly 1:1. This indicates that the lines of code metric is only suitable for predicting an upperbound on the factbase size.

The transformation of the metamodel into a graph schema in Chapter 5 meant factoring the associations into binary relations. This factorization is evident in the
number of facts for relations such as isLeftOperandOf and isRightOperandOf, and is-FormalParameterOf and isPositionOf. The numbers in both cases are identical since the relations encode what are essentially facets of higher arity relations. Also, approximately 60% of the facts for the application are for unary and binary expressions, such as those containing algebraic, logical, or assignment operators. This percentage is evenly divided among the three relations: isLeftOperandOf, isRightOperandOf, and isOperatorOf.

These observations indicate that it should be possible to reduce the sizes of the factbases by using higher arity relations. Graphs are useful for visualizing the relations, but are not strictly necessary to perform the analysis. Secondly, some relations could be partitioned. Rather than representing all binary expressions in one relation, they could be separated into one for assignments, one for algebraic expressions, and one for logical expressions. Partitioning does not reduce the overall size of the factbase, but does reduce the size of the relation being partitioned. Manipulations and queries that operate on only a subset of the relation that falls within a partition would benefit.

The second phase is the manipulation and pattern matching using CrocoPat. The RML programs are executed against the factbases by the automated scripts. The results are listed in Table 8.3 on the next page.

Each row shows the recovery results for an application based on the four implemented patterns. The results include: the number of parameters in the relevant (documented) set, the number of parameters in the retrieved (recovered) set, and the recall and precision computed from these two sets. There are several instances where an application did not have documented parameters of a certain type. The precision
Table 8.3: Results of the automated recovery for the applications.

<table>
<thead>
<tr>
<th></th>
<th>Toggle</th>
<th>Counter</th>
<th>Maximum</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Derby</td>
<td>6</td>
<td>71</td>
<td>67%</td>
<td>6%</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>19</td>
<td>163</td>
<td>84%</td>
<td>10%</td>
</tr>
<tr>
<td>Babylon Chat</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Hypersonic 2</td>
<td>3</td>
<td>106</td>
<td>100%</td>
<td>3%</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>8</td>
<td>48</td>
<td>100%</td>
<td>17%</td>
</tr>
<tr>
<td>Mort Bay Jetty</td>
<td>1</td>
<td>43</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Oracle Berkeley DB Java Edition</td>
<td>20</td>
<td>72</td>
<td>35%</td>
<td>10%</td>
</tr>
<tr>
<td><strong>AVERAGE</strong></td>
<td><strong>64%</strong></td>
<td><strong>7%</strong></td>
<td><strong>86%</strong></td>
<td><strong>22%</strong></td>
</tr>
</tbody>
</table>
Table 8.4: Percentage of undocumented parameters recovered that are considered spurious.

<table>
<thead>
<tr>
<th></th>
<th>Toggle</th>
<th>Counter</th>
<th>Maximum</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Derby</td>
<td>70%</td>
<td>40%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>30%</td>
<td>45%</td>
<td>60%</td>
<td>0%</td>
</tr>
<tr>
<td>Babylon Chat</td>
<td>-</td>
<td>0%</td>
<td>0%</td>
<td>-</td>
</tr>
<tr>
<td>Hypersonic 2</td>
<td>65%</td>
<td>40%</td>
<td>50%</td>
<td>-</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>65%</td>
<td>35%</td>
<td>0%</td>
<td>-</td>
</tr>
<tr>
<td>Mort Bay Jetty</td>
<td>30%</td>
<td>50%</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>Oracle Berkeley</td>
<td>25%</td>
<td>15%</td>
<td>0%</td>
<td>-</td>
</tr>
<tr>
<td>DB Java Edition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From an application perspective, the results show that parameters are successfully recovered from both the applications that were studied, as well as the others that were not previously studied. In fact, the recall and precision for both sets of applications are quite similar.

As mentioned above, the precisions in Table 8.3 on the previous page are actually better in practice. Manual validation of the undocumented parameters that are recovered reveals that many are still useful parameters for tuning. Those that are not useful are considered spurious. Table 8.4 lists the percentages of undocumented parameters recovered that are spurious.

Cases where the precision is already 100%, or where no tuning parameters are retrieved, are undefined. As the table indicates, none of the undocumented delays are spurious. For maximums, there are cases where none of the undocumented ones are spurious. Yet, there are also cases where all of the undocumented ones are spurious. The number of recovered parameters in these cases is low, so the fact that they are
spurious is not detrimental. In all cases for counters, less than half of the undocumented ones are spurious. Toggles have the poorest results. There are instances where more than half of the undocumented ones are spurious.

Deployment of the approach is meant to assist with the recovery of as many tuning parameters as possible. The emphasis is therefore on recall, and not precision. Additionally, it is important to note that the implementation not only returns what the parameters are, but also shows where they are. The parameters and their references can quickly be navigated. This allows the results to be more easily validated.

### 8.3 Summary

This chapter presented the evaluation results for an implementation of the tuning parameter recovery. The facts were automatically extracted using a plug-in that was developed for the Eclipse platform. These facts were written to files to be used as input to the CrocoPat tool. The fact manipulation and pattern matching were then performed using RML programs for CrocoPat.

The recovery was evaluated against the applications studied in Chapter 4, as well as three others. The purpose was to demonstrate that the approach could recover the documented tuning parameters of the studied applications, as well as others that were not previously studied. Two of these other applications were from the same domains as the studied applications, while the third was from a different domain. However, the patterns appeared to be agnostic to any particular application or domain.

Perhaps not suprisingly, the results indicated that not all tuning parameters are necessarily documented. However, the approach is effective at recovering both the documented tuning parameters, as well as these undocumented ones.
Chapter 9

Conclusion

Computing systems are becoming too complex to integrate and manage without some kind of automated assistance. Autonomic computing is a proposed solution to the complexity problem. To this end, the Software Tuning Panels for Autonomic Control project was started to assist with rearchitecting existing systems for autonomic control. The isolation of an application’s tuning parameters to a separate control panel has been automated using source transformation. However, the prerequisite tuning parameter identification step was done manually.

This thesis demonstrated the use of existing software analysis techniques for automatically recovering the tuning parameters. In doing so, it addressed the three questions posed in Chapter 1. The reverse-engineering community has developed techniques for reasoning about the roles of variables in source code. These involve parsing the source code to extract instances of relationships between source code elements, and analysing the relationships for patterns. To help identify the patterns, a tuning parameter classification was developed based on empirical study of tuning
parameter use in source code. It was evident from the study that not all tuning parameters are explicit variables. Some are, for example, implicit to the state of a data structure, while others are computed dynamically. It is surmised that these implicit tuning parameters can be made into explicit variables using source transformation.

The empirical study was conducted using several industrially relevant, server-oriented applications. The study looked at how the applications’ documented tuning parameters were manifested in source code. By mapping the parameters to field declarations and tracing references to the fields, patterns of use emerged. These patterns were used to classify the tuning parameters into a taxonomy. The characteristics of the references were used to develop a metamodel of source code elements and their relationships.

The relationships represented in the metamodel were extracted by first parsing the source code to generate abstract syntax trees. The abstract syntax trees were traversed by the extractor, which recognized instances of the relationships among the source code elements bound to the tree nodes. The relationship instances, called facts, formed binary relations (i.e., graphs) over the source code elements.

The extracted facts were manipulated to reduce the complexity of expressing the tuning parameter patterns. This helped to overcome the indirection induced by the object-oriented paradigm. The manipulations extended the existing relations, while also creating new, higher-level ones. To recover the tuning parameters, the manipulated facts were queried for instances of the patterns of use.

The recovery was implemented as a tool for finding tuning parameters in applications, and was evaluated against the applications from the empirical study, as well
as several others. Experimental results showed that the approach is effective at recovering documented tuning parameters, as well as other undocumented ones. The results also indicated that the tuning parameter patterns are not necessarily specific to a particular application, or application domain.

9.1 Limitations

While a number of applications have been studied, both large and small, mature and immature, the possibility remains that there are applications, and domains, for which this approach might not work. For example, does a graphical application have the same kinds of tuning parameters? Is tuning in that context vastly different from the server-oriented context?

More concretely, this work focused on object-oriented software, and in particular, Java. The metamodel and patterns are therefore biased toward source code written in the Java programming language. It can be argued that other languages, like C++, are amenable to this approach because of their similarity to Java. However, there is no experimental evidence to support such a claim. Nevertheless, the overall approach should still be applicable, albeit with a modified metamodel and patterns.

The tuning parameters studied are limited to explicitly declared scalar fields. While the vast majority of the documented tuning parameters fall within this scope, there are some that do not. For example, parameters whose values are computed dynamically are excluded. These parameters are manifested as methods that compute the value of the parameter each time the method is invoked, rather than storing it explicitly in a field.

Many source code element relationships are omitted from the metamodel to reduce
the time and space complexity of the analysis. For example, there are no relationships representing the relative ordering of expressions within statements, or between expressions in different statements. This omission makes the analysis flow-insensitive, and can result in over-approximations in flow-sensitive patterns, such as the one for statistical maximums.

9.2 Future Work

The integration of this work with the previous STAC prototype remains future work. As well, there are still a number of areas that could benefit from further research.

Not all of the tuning parameter patterns have been formally specified as RML programs and evaluated against the applications. While the ones that have been specified are thought to be a representative sample, there may be aspects of the others that require additional refinements to the metamodel to ensure an acceptable level of accuracy. As well, the RML programs that have been specified could be refined to improve their accuracy. This is especially true for those patterns that involve invocations of methods for which the source code is outside the scope of the application. For example, the precision of the counter pattern is improved by ignoring those fields which are used as array indices. However, there is an implicit array index relationship that is hidden by the `get(int)` method of the `Collection` classes in the Java API. By treating actual parameters to these methods as array indices, the precision of the pattern should improve. This could be done during the fact manipulation process by generating `isIndexOf` facts for expressions that are actual parameters of the `get(int)` methods.

The tuning parameter taxonomy could be improved based on expert feedback, and
study of a broader range of applications and domains. Also, the tuning parameters studied were limited to explicit scalar field declarations. Expanding this definition would mean defining new classifications. The size property of collections, for example, is accessible only through an accessor method. This property is, in a sense, an implicit tuning parameter. The concept of implicit tuning parameters can be extended to include properties such as the depth of tree data structures, or cache hit rates, for which even accessor methods may not exist.

The patterns used to identify the tuning parameters consider only how the parameters are referenced. It would be useful to develop an orthogonal classification based on what resources the parameters influence. For example, the toggle pattern has relatively few discerning characteristics in terms of how the field is referenced (i.e., whether it is a condition for some expression). The pattern therefore has low precision. However, knowing that the condition influences some resource (e.g., disables caching) would likely improve the chances of the recovered parameter being a relevant one.

The search plug-in implementation could be improved in a number of ways. There is no persistence of the factbases, other than that used for the evaluation. The factbases could be stored persistently, and updated only when the source code has been modified. This would reduce the overhead of extracting the facts each time a search is performed. Also, the fact manipulation and pattern matching are implemented in Java as naive binary relational operations. While convenient for this work, there are other, more mature tools designed for relational manipulation. Though they may require platform-specific binaries, they are far more efficient.
Finally, there are other aspects of tuning parameter recovery that should be explored. It would be useful to gather metadata about the parameters. For example, whether the parameter is mutable, whether it is thread-safe, and if it has a default value, a maximum value, or a minimum value.
Bibliography


Appendix A

Tuning Parameters and Classifications

The tables in the following sections list the documented tuning parameters, and their classifications, for the applications studied in Chapter 4.
### A.1 Apache Derby

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>derby.locks.deadlockTimeout</td>
<td>org.apache.derby.impl.services.locks.LockSet.deadlockTimeout</td>
<td>Timeout</td>
</tr>
<tr>
<td>derby.locks.deadlockTrace</td>
<td>org.apache.derby.impl.services.locks.LockSet.deadlockTrace</td>
<td>Toggle</td>
</tr>
<tr>
<td>derby.locks.escalationThreshold</td>
<td>org.apache.derby.impl.services.locks.LockSpace.limit</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.locks.monitor</td>
<td>org.apache.derby.iapi.error.StandardException.report</td>
<td>Enumerable</td>
</tr>
<tr>
<td>derby.locks.waitTimeout</td>
<td>org.apache.derby.impl.services.locks.LockSet.waitTimeout</td>
<td>Timeout</td>
</tr>
<tr>
<td>derby.storage.initialPages</td>
<td>org.apache.derby.impl.store.raw.data.FileContainer.preAllocSize</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.storage.minimumRecordSize</td>
<td>org.apache.derby.impl.store.raw.data.StoredPage.minimumRecordSize</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.storage.pageCacheSize</td>
<td>org.apache.derby.impl.services.cache.Clock.maximumSize</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.storage.pageReservedSpace</td>
<td>org.apache.derby.impl.store.raw.data.StoredPage.spareSpace</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.storage.pageSize</td>
<td>org.apache.derby.impl.store.raw.data.FileContainer.pageSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>derby.stream.error.logSeverityLevel</td>
<td>org.apache.derby.iapi.services.context.ContextManager.logSeverityLevel</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.drda.logConnections</td>
<td>org.apache.derby.impl.drda.DRDAConnThread.logConnections</td>
<td>Toggle</td>
</tr>
<tr>
<td>derby.drda.maxThreads</td>
<td>org.apache.derby.impl.drda.NetworkServerControlImpl.maxThreads</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.drda.minThreads</td>
<td>org.apache.derby.impl.drda.NetworkServerControlImpl.minThreads</td>
<td>Threshold</td>
</tr>
<tr>
<td>derby.drda.timeSlice</td>
<td>org.apache.derby.impl.drda.ClientThread.timeSlice</td>
<td>Timeout</td>
</tr>
<tr>
<td>derby.drda.traceAll</td>
<td>org.apache.derby.impl.drda.NetworkServerControlImpl.traceAll</td>
<td>Toggle</td>
</tr>
</tbody>
</table>
## A.2 Apache Tomcat

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
</table>

### Parameter Name

- `org.apache.catalina.ha.session.DeltaManager.counterSend_EVT_SESSION_EXPIRED`
- `org.apache.catalina.ha.session.DeltaManager.counterSend_EVT_SESSION_DELTA`
- `org.apache.catalina.ha.session.DeltaManager.counterSend_EVT_SESSION_CREATED`
- `org.apache.catalina.ha.session.DeltaManager.counterSend_EVT_SESSION_ACCESSED`
- `org.apache.catalina.ha.session.DeltaManager.counterSend_EVT_GET_ALL_SESSIONS`
- `org.apache.catalina.ha.session.DeltaManager.counterSend_EVT_ALL_SESSION_DATA`
- `org.apache.catalina.ha.session.DeltaManager.counterReceive_EVT_SESSION_EXPIRED`
- `org.apache.catalina.ha.session.DeltaManager.counterReceive_EVT_SESSION_DELTA`
- `org.apache.catalina.ha.session.DeltaManager.counterReceive_EVT_SESSION_CREATED`
- `org.apache.catalina.ha.session.DeltaManager.counterReceive_EVT_SESSION_ACCESSED`
- `org.apache.catalina.ha.session.DeltaManager.counterReceive_EVT_GET_ALL_SESSIONS`
- `org.apache.catalina.ha.session.DeltaManager.counterReceive_EVT_ALL_SESSION_TRANSFERCOMPLETE`
- `org.apache.catalina.ha.session.DeltaManager.counterNoStateTransfered`
- `org.apache.catalina.core.StandardWrapper.requestCount`
- `org.apache.catalina.core.StandardWrapper.processingTime`
- `org.apache.catalina.core.StandardWrapper.maxTime`
- `org.apache.catalina.core.StandardWrapper.loadTime`
- `org.apache.catalina.core.StandardWrapper.classLoadTime`
- `org.apache.catalina.core.StandardContext.unloadDelay`
- `org.apache.catalina.core.StandardContext.tldScanTime`
- `org.apache.catalina.core.StandardContext.startupTime`
- `org.apache.catalina.core.StandardContext.startTime`
- `org.apache.catalina.core.StandardContext.cacheTTL`
- `org.apache.catalina.connector.Connector.maxSpareThreads`
- `org.apache.catalina.connector.Connector.maxPostSize`
- `org.apache.catalina.connector.Connector.maxHttpHeaderSize`
- `org.apache.catalina.connector.Connector.keepAliveTimeout`
- `org.apache.catalina.connector.Connector.enableLookups`
- `org.apache.catalina.connector.Connector.connectionTimeout`
- `org.apache.catalina.connector.Connector.connectionLinger`
- `org.apache.catalina.connector.Connector.allowTrace`
- `org.apache.catalina.connector.Connector.acceptCount`
- `org.apache.catalina.cluster.authenticator.ClusterSingleSignOn.requireReauthentication`
- `org.apache.catalina.authenticator.NonLoginAuthenticator.cache`
- `org.apache.catalina.authenticator.DigestAuthenticator.cache`
- `org.apache.catalina.authenticator.BasicAuthenticator.cache`

### Field Declaration


### Classification

- `Counter`
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.apache.catalina.ha.session.DeltaManager.duplicates</td>
<td>org.apache.catalina.session.ManagerBase.duplicates</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.session.StandardManager.duplicates</td>
<td>org.apache.catalina.session.ManagerBase.duplicates</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.ha.session.DeltaManager.expiredSessions</td>
<td>org.apache.catalina.session.ManagerBase.expiredSessions</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.session.PersistentManager.expiredSessions</td>
<td>org.apache.catalina.session.ManagerBase.expiredSessions</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.ha.session.DeltaManager.maxActive</td>
<td>org.apache.catalina.session.ManagerBase.maxActive</td>
<td>Maximum</td>
</tr>
<tr>
<td>org.apache.catalina.session.PersistentManager.maxActive</td>
<td>org.apache.catalina.session.ManagerBase.maxActive</td>
<td>Maximum</td>
</tr>
<tr>
<td>org.apache.catalina.ha.session.DeltaManager.maxActiveSessions</td>
<td>org.apache.catalina.session.ManagerBase.maxActiveSessions</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.apache.catalina.session.PersistentManager.maxActiveSessions</td>
<td>org.apache.catalina.session.ManagerBase.maxActiveSessions</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.apache.catalina.ha.session.DeltaManager.processExpiredFrequency</td>
<td>org.apache.catalina.session.ManagerBase.processExpiredFrequency</td>
<td>Frequency</td>
</tr>
<tr>
<td>org.apache.catalina.session.PersistentManager.processExpiredFrequency</td>
<td>org.apache.catalina.session.ManagerBase.processExpiredFrequency</td>
<td>Frequency</td>
</tr>
<tr>
<td>org.apache.catalina.session.StandardManager.processExpiredFrequency</td>
<td>org.apache.catalina.session.ManagerBase.processExpiredFrequency</td>
<td>Frequency</td>
</tr>
<tr>
<td>org.apache.catalina.ha.tcp.ReplicationValve.totalSendTime</td>
<td>org.apache.catalina.ha.tcp.ReplicationValve.totalSendTime</td>
<td>Duration</td>
</tr>
<tr>
<td>org.apache.catalina.ha.tcp.ReplicationValve.totalRequestTime</td>
<td>org.apache.catalina.ha.tcp.ReplicationValve.totalRequestTime</td>
<td>Duration</td>
</tr>
<tr>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfSendRequests</td>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfSendRequests</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfRequests</td>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfRequests</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfFilterRequests</td>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfFilterRequests</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfCrossContextSendRequests</td>
<td>org.apache.catalina.ha.tcp.ReplicationValve.nrOfCrossContextSendRequests</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.ha.tcp.ReplicationValve.lastSendTime</td>
<td>org.apache.catalina.ha.tcp.ReplicationValve.lastSendTime</td>
<td>Point</td>
</tr>
<tr>
<td>org.apache.catalina.ha.session.PersistentManager.maxIdleSwap</td>
<td>org.apache.catalina.session.PersistentManagerBase.maxIdleSwap</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.apache.catalina.ha.session.PersistentManager.maxActiveSessions</td>
<td>org.apache.catalina.session.PersistentManagerBase.maxActiveSessions</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.apache.catalina.ha.session.StandardManager.processingTime</td>
<td>org.apache.catalina.session.ManagerBase.processingTime</td>
<td>Duration</td>
</tr>
<tr>
<td>org.apache.catalina.session.StandardManager.rejectedSessions</td>
<td>org.apache.catalina.session.ManagerBase.rejectedSessions</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.catalina.session.valvesSemaphoreValves.concurrency</td>
<td>org.apache.catalina.valvesSemaphoreValves.concurrency</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.apache.catalina.session.valvesSemaphoreValves.size</td>
<td>org.apache.catalina.valvesSemaphoreValves.size</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.maxSpamThreads</td>
<td>org.apache.tomcat.util.threads.ThreadPool.maxSpamThreads</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.minThreads</td>
<td>org.apache.tomcat.util.threads.ThreadPool.minThreads</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.maxSpamThreads</td>
<td>org.apache.tomcat.util.threads.ThreadPool.maxSpamThreads</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.minThreads</td>
<td>org.apache.tomcat.util.threads.ThreadPool.minThreads</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.queueSize</td>
<td>org.apache.jk.common.ChannelSocket.queueSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.requestCount</td>
<td>org.apache.jk.common.ChannelSocket.requestCount</td>
<td>Counter</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.scheduled</td>
<td>org.apache.jk.common.ChannelSocket.scheduled</td>
<td>Delay</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.socketTimeout</td>
<td>org.apache.jk.common.ChannelSocket.socketTimeout</td>
<td>Timeout</td>
</tr>
<tr>
<td>org.apache.jk.common.ChannelSocket.tcpNoDelay</td>
<td>org.apache.jk.common.ChannelSocket.tcpNoDelay</td>
<td>Toggle</td>
</tr>
</tbody>
</table>
A.3 Mort Bay Jetty

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContextDeployer.scanInterval</td>
<td>org.mortbay.util.Scanner._scanInterval</td>
<td>Period</td>
</tr>
<tr>
<td>StatisticsHandler.requests</td>
<td>org.mortbay.jetty.handler.StatisticsHandler._requests</td>
<td>Counter</td>
</tr>
<tr>
<td>StatisticsHandler.requestsActive</td>
<td>org.mortbay.jetty.handler.StatisticsHandler._requestsActive</td>
<td>Counter</td>
</tr>
<tr>
<td>StatisticsHandler.requestsActiveMin</td>
<td>org.mortbay.jetty.handler.StatisticsHandler._requestsActiveMin</td>
<td>Minimum</td>
</tr>
<tr>
<td>StatisticsHandler.requestsActiveMax</td>
<td>org.mortbay.jetty.handler.StatisticsHandler._requestsActiveMax</td>
<td>Maximum</td>
</tr>
<tr>
<td>StatisticsHandler.requestsDurationMin</td>
<td>org.mortbay.jetty.handler.StatisticsHandler._requestsDurationMin</td>
<td>Minimum</td>
</tr>
<tr>
<td>StatisticsHandler.requestsDurationMax</td>
<td>org.mortbay.jetty.handler.StatisticsHandler._requestsDurationMax</td>
<td>Maximum</td>
</tr>
<tr>
<td>StatisticsHandler.requestsDurationTotal</td>
<td>org.mortbay.jetty.handler.StatisticsHandler._requestsDurationTotal</td>
<td>Duration</td>
</tr>
<tr>
<td>Connector.headerBufferSize</td>
<td>org.mortbay.jetty.AbstractBuffers._headerBufferSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>Connector.requestBufferSize</td>
<td>org.mortbay.jetty.AbstractBuffers._requestBufferSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>Connector.responseBufferSize</td>
<td>org.mortbay.jetty.AbstractBuffers._responseBufferSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>Connector.maxIdleTime</td>
<td>org.mortbay.jetty.AbstractConnector._maxIdleTime</td>
<td>Timeout</td>
</tr>
<tr>
<td>Connector.connections</td>
<td>org.mortbay.jetty.AbstractConnector._connections</td>
<td>Counter</td>
</tr>
<tr>
<td>Connector.connectionsOpen</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsOpen</td>
<td>Counter</td>
</tr>
<tr>
<td>Connector.connectionsOpenMin</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsOpenMin</td>
<td>Minimum</td>
</tr>
<tr>
<td>Connector.connectionsOpenMax</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsOpenMax</td>
<td>Maximum</td>
</tr>
<tr>
<td>Connector.connectionsDurationMin</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsDurationMin</td>
<td>Minimum</td>
</tr>
<tr>
<td>Connector.connectionsDurationMax</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsDurationMax</td>
<td>Maximum</td>
</tr>
<tr>
<td>Connector.connectionsDurationTotal</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsDurationTotal</td>
<td>Duration</td>
</tr>
<tr>
<td>Connector.connectionsRequestsMin</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsRequestsMin</td>
<td>Minimum</td>
</tr>
<tr>
<td>Connector.connectionsRequestsMax</td>
<td>org.mortbay.jetty.AbstractConnector._connectionsRequestsMax</td>
<td>Maximum</td>
</tr>
<tr>
<td>Connector.requests</td>
<td>org.mortbay.jetty.AbstractConnector._requests</td>
<td>Accumulator</td>
</tr>
<tr>
<td>Server.startupTime</td>
<td>org.mortbay.jetty.management.ServerMBean.startupTime</td>
<td>Point</td>
</tr>
<tr>
<td>SelectChannelConnector.lowResourcesConnections</td>
<td>org.mortbay.jetty.nio.SelectChannelConnector._lowResourcesConnections</td>
<td>Threshold</td>
</tr>
<tr>
<td>AbstractSessionManager.maxSessions</td>
<td>org.mortbay.jetty.servlet.AbstractSessionManager._maxSessions</td>
<td>Maximum</td>
</tr>
<tr>
<td>AbstractSessionManager.minSessions</td>
<td>org.mortbay.jetty.servlet.AbstractSessionManager._minSessions</td>
<td>Minimum</td>
</tr>
<tr>
<td>AbstractSessionManager.refreshCookieAge</td>
<td>org.mortbay.jetty.servlet.AbstractSessionManager._refreshCookieAge</td>
<td>Timeout</td>
</tr>
<tr>
<td>AbstractSessionManager.secureCookies</td>
<td>org.mortbay.jetty.servlet.AbstractSessionManager._secureCookies</td>
<td>Toggle</td>
</tr>
<tr>
<td>BoundedThreadPool.minThreads</td>
<td>org.mortbay.thread.BoundedThreadPool._minThreads</td>
<td>Threshold</td>
</tr>
<tr>
<td>BoundedThreadPool.maxThreads</td>
<td>org.mortbay.thread.BoundedThreadPool._maxThreads</td>
<td>Threshold</td>
</tr>
<tr>
<td>BoundedThreadPool.threadsPriority</td>
<td>org.mortbay.thread.BoundedThreadPool._priority</td>
<td>Enumeration</td>
</tr>
<tr>
<td>BoundedThreadPool.maxIdleTimeMs</td>
<td>org.mortbay.thread.BoundedThreadPool._maxIdleTimeMs</td>
<td>Timeout</td>
</tr>
</tbody>
</table>
## A.4 Oracle Berkeley DB Java Edition

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>je.adler32.chunkSize</td>
<td>com.sleepycat.je.util.Adler32.adler32ChunkSize</td>
<td>Threshold</td>
</tr>
<tr>
<td>je.checkpointer.bytesInterval</td>
<td>com.sleepycat.je.util.Checkpointer.logSizeByBytesInterval</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.checkpointer.lruBatchSize</td>
<td>com.sleepycat.je.util.Checkpointer.lruBatchSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.checkpointTimeout</td>
<td>com.sleepycat.je.util.CheckpointManager.timeoutMillis</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.cluster</td>
<td>com.sleepycat.je.cleaner.Cleaner.cluster</td>
<td>Timeout</td>
</tr>
<tr>
<td>je.cleaner.clusterAll</td>
<td>com.sleepycat.je.cleaner.Cleaner.clusterAll</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.deadlockRetry</td>
<td>com.sleepycat.je.cleaner.Cleaner.nDeadlockRetries</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.expunge</td>
<td>com.sleepycat.je.cleaner.Cleaner.expunge</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.fetchObsoleteSize</td>
<td>com.sleepycat.je.cleaner.Cleaner.fetchObsoleteSize</td>
<td>Timeout</td>
</tr>
<tr>
<td>je.cleaner.lockManagerTimeout</td>
<td>com.sleepycat.je.txn.LockManager.lockTableDumpTimeout</td>
<td>Timeout</td>
</tr>
<tr>
<td>je.cleaner.lockManagerTimeoutCacheSize</td>
<td>com.sleepycat.je.cleaner.Cleaner.lockManagerTimeoutCacheSize</td>
<td>Timeout</td>
</tr>
<tr>
<td>je.cleaner.lockManagerTimeoutCacheThreshold</td>
<td>com.sleepycat.je.cleaner.Cleaner.lockManagerTimeoutCacheThreshold</td>
<td>Timeout</td>
</tr>
<tr>
<td>je.cleaner.maxBatchFiles</td>
<td>com.sleepycat.je.cleaner.Cleaner.maxBatchFiles</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.minAge</td>
<td>com.sleepycat.je.cleaner.Cleaner.minAge</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.minFileUtilization</td>
<td>com.sleepycat.je.cleaner.Cleaner.minFileUtilization</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.readSize</td>
<td>com.sleepycat.je.cleaner.Cleaner.readBufferSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.cleaner.threads</td>
<td>com.sleepycat.je.cleaner.Cleaner.threads</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.compressor.wakeupInterval</td>
<td>com.sleepycat.je.util.DaemonThread.waitTime</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.compressor.wakeupInterval</td>
<td>com.sleepycat.je.util.DaemonThread.waitTime</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.reflectWriteTemp</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.reflectWriteTemp</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.env.backgroundReadLimit</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.backgroundReadLimit</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.env.backgroundSleepInterval</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.backgroundSleepInterval</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.env.isTransactional</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.isTransactional</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.env.sharedLatches</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.sharedLatches</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.evictor.evictByLruOnly</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.evictByLruOnly</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.evictor.lruOnly</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.evictByLruOnly</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.evictor.nodesPerScan</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.evictNodesPerScan</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.lockManager.lruTimeout</td>
<td>com.sleepycat.je.dbi.LockManager.lruTimeout</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.lockManager.lruTimeout</td>
<td>com.sleepycat.je.dbi.LockManager.lruTimeout</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.checksumRead</td>
<td>com.sleepycat.je.log.FileManager.checksumOnRead</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.chunkedNO</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.chunkedNO</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.directWrite</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.directWrite</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.fileCacheSize</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.fileCacheSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.fileMax</td>
<td>com.sleepycat.je.dbi.EnvironmentImpl.fileMax</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.syncTimeout</td>
<td>com.sleepycat.je.log.Checkpointer.logSyncTimeout</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.iteratorMaxSize</td>
<td>com.sleepycat.je.log.FileManager.iteratorMaxSize</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.log.useNIO</td>
<td>com.sleepycat.je.log.FileManager.useNIO</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.nodeDeadlockEntries</td>
<td>com.sleepycat.je.dbi.TreeImpl.nodeDeadlockEntries</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.nodeDeadlockEntries</td>
<td>com.sleepycat.je.dbi.TreeImpl.nodeDeadlockEntries</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.tree.binDelta</td>
<td>com.sleepycat.je.dbi.DatabaseImpl.binDelta</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.tree.maxDelta</td>
<td>com.sleepycat.je.dbi.DatabaseImpl.maxDelta</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.txn.deadlockStackTrace</td>
<td>com.sleepycat.je.dbi.LockManager.deadlockStackTrace</td>
<td>Capacity</td>
</tr>
<tr>
<td>je.txn.timeout</td>
<td>com.sleepycat.je.dbi.LockManager.timeout</td>
<td>Capacity</td>
</tr>
</tbody>
</table>

---

APPENDIX A. TUNING PARAMETERS AND CLASSIFICATIONS

152
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.sleepycat.je.EnvironmentStats.nWaits</td>
<td>com.sleepycat.je.EnvironmentStats.nWaits</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.EnvironmentStats.nRequests</td>
<td>com.sleepycat.je.EnvironmentStats.nRequests</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.TransactionStats.nAborts</td>
<td>com.sleepycat.je.TransactionStats.nAborts</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.TransactionStats.nCommits</td>
<td>com.sleepycat.je.TransactionStats.nCommits</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.TransactionStats.nXAAborts</td>
<td>com.sleepycat.je.TransactionStats.nXAAborts</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.TransactionStats.nXACommit</td>
<td>com.sleepycat.je.TransactionStats.nXACommit</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.TransactionStats.nXACommits</td>
<td>com.sleepycat.je.TransactionStats.nXACommits</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.LockStats.nWaits</td>
<td>com.sleepycat.je.LockStats.nWaits</td>
<td>Counter</td>
</tr>
<tr>
<td>com.sleepycat.je.LockStats.nRequests</td>
<td>com.sleepycat.je.LockStats.nRequests</td>
<td>Counter</td>
</tr>
</tbody>
</table>
Appendix B

Tuning Parameters and Classifications

The tables in the following sections list the documented tuning parameters, and their classifications, for the additional applications evaluated in Chapter 8.
## B.1 Hypersonic 2

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOW_LITERALS</td>
<td>org.h2.engine.Database.allowLiterals</td>
<td>Enumerable</td>
</tr>
<tr>
<td>CACHE_SIZE</td>
<td>org.h2.util.Cache2Q.maxSize</td>
<td>Threshold</td>
</tr>
<tr>
<td></td>
<td>org.h2.util.CacheLRU.maxSize</td>
<td></td>
</tr>
<tr>
<td>DB_CLOSE_DELAY</td>
<td>org.h2.engine.DatabaseCloser.delayInMillis</td>
<td>Delay</td>
</tr>
<tr>
<td>LOCK_MODE</td>
<td>org.h2.engine.Database.lockMode</td>
<td>Enumerable</td>
</tr>
<tr>
<td>LOCK_TIMEOUT</td>
<td>org.h2.engine.Session.lockTimeout</td>
<td>Timeout</td>
</tr>
<tr>
<td>LOG</td>
<td>org.h2.engine.Database.logLevel</td>
<td></td>
</tr>
<tr>
<td>MAX_LENGTH_INPLACE_LOB</td>
<td>org.h2.engine.Database.maxLengthInplaceLob</td>
<td>Threshold</td>
</tr>
<tr>
<td>MAX_LOG_SIZE</td>
<td>org.h2.log.LogSystem.maxLogSize</td>
<td>Threshold</td>
</tr>
<tr>
<td>MAX_MEMORY_ROWS</td>
<td>org.h2.result.LocalResult.maxMemoryRows</td>
<td>Threshold</td>
</tr>
<tr>
<td>MAX_MEMORY_UNDO</td>
<td>org.h2.engine.Database.maxMemoryUndo</td>
<td>Threshold</td>
</tr>
<tr>
<td>OPTIMIZE_REUSE_RESULTS</td>
<td>org.h2.engine.Database.optimizeReuseResults</td>
<td>Toggle</td>
</tr>
<tr>
<td>REFERENTIAL_INTEGRITY</td>
<td>org.h2.engine.Database.referentialIntegrity</td>
<td>Toggle</td>
</tr>
<tr>
<td>THROTTLE</td>
<td>org.h2.engine.Session.throttle</td>
<td>Delay</td>
</tr>
<tr>
<td>TRACE_LEVEL_FILE</td>
<td>org.h2.message.TraceSystem.levelFile</td>
<td>Enumerable</td>
</tr>
<tr>
<td>TRACE_LEVEL_SYSTEM_OUT</td>
<td>org.h2.message.TraceSystem.levelSystemOut</td>
<td>Enumerable</td>
</tr>
<tr>
<td>TRACE_MAX_FILE_SIZE</td>
<td>org.h2.message.TraceSystem.maxFileSize</td>
<td>Threshold</td>
</tr>
<tr>
<td>UNDO_LOG</td>
<td>org.h2.engine.Session.undoLogEnabled</td>
<td>Toggle</td>
</tr>
<tr>
<td>WRITE_DELAY</td>
<td>org.h2.store.WriterThread.writeDelay</td>
<td>Delay</td>
</tr>
<tr>
<td>info.FILE_DISK_WRITE</td>
<td>org.h2.store.DiskFile.writeCount</td>
<td>Counter</td>
</tr>
<tr>
<td>info.FILE_INDEX_WRITE</td>
<td>org.h2.store.DiskFile.readCount</td>
<td></td>
</tr>
<tr>
<td>info.FILE_DISK_READ</td>
<td>org.h2.store.DiskFile.readCount</td>
<td></td>
</tr>
<tr>
<td>info.FILE_INDEX_READ</td>
<td>org.h2.store.DiskFile.readCount</td>
<td></td>
</tr>
</tbody>
</table>
## APPENDIX B. TUNING PARAMETERS AND CLASSIFICATIONS

### B.2 Jigsaw

<table>
<thead>
<tr>
<th>Tuning Parameter</th>
<th>Field Declaration</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.w3c.jigsaw.keepAlive</td>
<td>org.w3c.jigsaw.httpd.keep</td>
<td>Toggle</td>
</tr>
<tr>
<td>org.w3c.jigsaw.client.priority</td>
<td>org.w3c.jigsaw.httpd.client_priority</td>
<td>Enumerable</td>
</tr>
<tr>
<td>org.w3c.jigsaw.client.bufsize</td>
<td>org.w3c.jigsaw.httpd.client_bufsize</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.w3c.jigsaw.client.debug</td>
<td>org.w3c.jigsaw.httpd.Client.debug</td>
<td>Toggle</td>
</tr>
<tr>
<td>org.w3c.jigsaw.request.timeout</td>
<td>org.w3c.jigsaw.httpd.request_timeout</td>
<td>Timeout</td>
</tr>
<tr>
<td>org.w3c.jigsaw.http.socket.SocketClientFactory.minFree</td>
<td>org.w3c.jigsaw.httpd.SocketClientFactory.minFree</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.w3c.jigsaw.http.socket.SocketClientFactory.maxFree</td>
<td>org.w3c.jigsaw.httpd.SocketClientFactory.maxFree</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.w3c.jigsaw.http.socket.SocketClientFactory.maxIdle</td>
<td>org.w3c.jigsaw.httpd.SocketClientFactory.maxIdle</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.w3c.jigsaw.http.socket.SocketClientFactory.maxClients</td>
<td>org.w3c.jigsaw.httpd.SocketClientFactory.maxClients</td>
<td>Capacity</td>
</tr>
<tr>
<td>org.w3c.jigsaw.httpd.FileCache.cache.size</td>
<td>org.w3c.jigsaw.httpd.FileCache.bytelimit</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.w3c.jigsaw.httpd.FileCache.debug</td>
<td>org.w3c.jigsaw.httpd.FileCache.debug</td>
<td>Toggle</td>
</tr>
<tr>
<td>org.w3c.jigsaw.httpd.FileCache.shared</td>
<td>org.w3c.jigsaw.httpd.FileCache.shared</td>
<td>Toggle</td>
</tr>
<tr>
<td>org.w3c.jigsaw.servlet.max-sessions</td>
<td>org.w3c.jigsaw.servlet.JigsawHttpSessionContext.maxsession</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.w3c.jigsaw.servlet.sessions-max-idle-time</td>
<td>org.w3c.jigsaw.servlet.JigsawHttpSessionContext.maxidle</td>
<td>Threshold</td>
</tr>
<tr>
<td>org.w3c.jigsaw.servlet.sessions-sweep-delay</td>
<td>org.w3c.jigsaw.servlet.JigsawHttpSessionContext.SweepDelay</td>
<td>Delay</td>
</tr>
</tbody>
</table>