UNIVERSITY OF CALGARY

Lightweight Support for Estimation of Polylingual Dependencies

by

Bradley E. Cossette

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
November, 2008

© Bradley E. Cossette 2008
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Lightweight Support for Estimation of Polylingual Dependencies” submitted by Bradley E. Cossette in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Dr. Robert J. Walker, Supervisor / Examining Committee Member
Department of Computer Science

Dr. Günter Ruhe, Co-Supervisor / Acting Supervisor
Department of Computer Science

Dr. Jörg Denzinger, Examining Committee Member
Department of Computer Science

Dr. Elizabeth Ritter, Examining Committee Member
Department of Linguistics

Date
Abstract

Software dependency analysis is an important step in determining the potential impact of changes to existing systems. Existing tool support for conducting dependency analysis does not sufficiently support systems written in more than one language. Tools based on semantic analysis are expensive to create for combinations of multiple languages, while lexical tools provide poor accuracy and rely heavily on developer skill.

This work proposes lightweight, approximate, polylingual dependency analysis tool support, by presuming which few semantics are important for such analysis, and requiring that developers only specify those syntactic patterns necessary to recognize these semantics. This work presents two studies conducted to evaluate (1) which semantics are of greatest use for such analysis, and (2) the ease-of-configuration, and effectiveness of such an approach, as implemented in the research prototype tool, GrammarSketch.
Acknowledgements

I was thrown out of college for cheating. Well, it was with the dean’s wife.

Woody Allen

In the summer of 1996, I was talking with a friend of mine—David Lee—about how my college education was going. I was not keen on chemical engineering anymore, but I was already two years in, and figured my “C’s get degrees” pace would get me out in another two or three years. Dave suggested I take a look at into Computer Science instead. That one, stupid statement, has started a chain reaction that’s lead to me floating around in “General Studies” limbo because I could not just go directly into Computer Science, getting kicked out for poor academic performance, getting readmitted (and this time, directly into CS), graduating with distinction, deciding to go into grad school, talking to my supervisor two days before a paper publication deadline as an absolutely broken man because I had spent the entire week writing and rewriting one chapter that every time he read it, he would send it back to me with the comment “not right, but I don’t know what right is”, being bolstered into giving that chapter one more try, submitting that paper at 3 a.m. on my birthday, getting a cryptic e-mail from Rob telling me the paper got accepted, traveling to Paris to present the paper, trying to keep a poker face during my talk while I mentally scream as my slides start automatically advancing on me, getting an NSERC for my Masters program after having been rejected twice for the award, wrestling with deciding to do the PhD, getting rejected from an Open Scholarship, applying for admission to the PhD anyways, getting an NSERC for the PhD, scrambling like mad to turn in that PhD acceptance letter, watching a case study for my Masters thesis start off in disaster, writing the thesis, still writing it two months after I had planned to be done, and finally, the smile on the face of my examination committee chair as the sign that this part was all over.

Thank you Dave, wherever you are—I could have been an engineer with a mind-numbingly boring, and well paying career a long time ago. Now, I actually think this stuff is fun!
There are a lot of people I need to thank for helping me to get to this point. Robert J. Walker, my supervisor, has put up with more from myself and my lab-mates than should ever be expected from a human being; at times, he is the patient teacher with the “special needs” student, the cheerleader for the doubting, the taskmaster for the flagging, and the court jester for the despairing. Güenther Ruhe and one of his graduate students (Pankaj Bhawnani) are also responsible for my decision to enter into graduate school. Güenther gave me the initial exposure to what academic research was like, and Pankaj spent an entire evening years ago extolling the value of a graduate education. Thank you Pankaj: three years later, and I’m still sold.

For my lab-mates—“Sweet, sweet” Rylan Cottrell, “Big” Joe Chang, the lab raconteur—Puneet “Butter Chicken isn’t an Indian dish!” Kapur, Bhavya “Red Van” Rawal, Reid “Water Bottle” Holmes, who was often our defacto supervisor, and Craig Schock—Reid and Craig both set the example we all tried to imitate—thank you for making these memorable years. I think the raucous nature of our humour reflects the stresses we’ve all been through, and I still cannot believe the good fortune I had to end up in a lab with so many other people who love to cook, and love to laugh.

Thank you to my family—Bernard and Diana, Aaron and Alicia, Brandon and Chrissy—for your support and prayers over the years, especially as I tried to wrap my head around going beyond a Masters degree. Thanks also go to the members of my “Home Group” at Bethany Chapel, who have prayed with and for me (especially during the last three weeks of the thesis writing)—Jim and Rosemary, Dan and Carol, Clint, DJ, Rob, Liesa, and Justyne—thank you all for your encouragement, and for bringing me back down to earth on so many occasions. Finally, I want to thank my God, and my Saviour Jesus Christ, who have been “at my right hand” in all of this (Psalm 16:8). My graduate studies have paralleled a different education I am also undergoing on faith, religion, spirituality, and the recovery of a heavily damaged relationship with God. Like grad school, I have decided to keep going with it.
For my parents, Bernard and Diana Cossette,
who always impressed on me the value of education.
Table of Contents

Approval Page ............................................................... ii
Abstract .............................................................................. iii
Acknowledgements ............................................................ iv
Dedication ........................................................................... vi
Table of Contents ............................................................... vii
List of Tables ........................................................................ xi
List of Figures ....................................................................... xii

1 Introduction .................................................................. 1
1.1 Background: Evolving Software Systems ......................... 3
  1.1.1 The Role of Dependency Analysis ............................... 3
  1.1.2 The Problem of Polylingual Software Systems ............. 5
1.2 Overview of Related Work ............................................. 5
1.3 The GrammarSketch Tool .............................................. 6
1.4 Thesis Statement .......................................................... 7
1.5 Overview of Thesis ...................................................... 8

2 Motivation ................................................................. 9
2.1 Scenario: Modifying the JPetStore Web-Based System ......... 9
  2.1.1 JPetStore: An Architecture and Language Overview ...... 10
2.2 Conducting a Dependency Analysis on JPetStore ................ 11
  2.2.1 The Relationship Between the Java and the JSP SubSystems . 13
  2.2.2 How Can This Go So Wrong? ...................................... 15
2.3 Regular Expression Based Lexical Analysis ......................... 16
  2.3.1 Example: Defining a Basic Identifier Pattern ............... 17
  2.3.2 Example: Expanding the Pattern for the Method Name .... 17
  2.3.3 Example: Adding Recognition of Parameter Lists .......... 18
  2.3.4 Example: Dealing With Exceptions in the Method Declaration 19
  2.3.5 Example: Limitations of the Pattern ......................... 20
  2.3.6 Summary of Example .............................................. 20
2.4 Grammar Based Syntactic Analysis .................................. 21
2.5 Summary ...................................................................... 22

3 Related Work ............................................................. 24
3.1 Introduction ............................................................... 24
3.2 Dependency and Impact Analysis .................................... 25
3.3 Lexical Approaches to Dependency Analysis ..................... 25
  3.3.1 The Lexical Source Model Extraction (LSME) Tool ....... 25
  3.3.2 The TAWK Tool .................................................. 27
3.4 Syntactic Approaches to Polylingual Dependency Analysis .... 27
  3.4.1 Interactions between Java and C Using the JNI Technology 28
  3.4.2 Interactions Between Java and SQL Using the JDBC Library 31
3.5 Island Grammars: A Hybrid Approach to Dependency Analysis 32
3.6 Summary ...................................................................... 33
4 The Luther Testbed .................................................. 34
  4.1 Overview ...................................................... 34
    4.1.1 Architectural Overview ................................. 35
  4.2 Configuring the Luther Testbed ............................ 36
    4.2.1 Defining Token Attributes ............................ 36
    4.2.2 Defining Dependency Rules ........................... 37
    4.2.3 Defining Island Grammars ............................ 37
  4.3 Experimental Evaluation ................................... 39
    4.3.1 Methodology ............................................ 40
    4.3.2 Island Grammar Sets .................................. 42
  4.4 Results ....................................................... 45
  4.5 Discussion .................................................. 47
    4.5.1 Limitations of the Study .............................. 47
    4.5.2 Future Work ............................................ 48
  4.6 Summary ..................................................... 49
5 The GrammarSketch Tool ......................................... 50
  5.1 Introduction ................................................ 50
    5.1.1 Lessons Learned From Luther .......................... 51
    5.1.2 Shifting From Island Grammars to Regular Expressions 52
  5.2 The GrammarSketch Tool ..................................... 53
    5.2.1 The Architecture of GrammarSketch .................... 54
    5.2.2 Simplifying Configuration Through Imposition ........ 56
  5.3 The GrammarSketch Notation ................................ 56
    5.3.1 Principles of the GrammarSketch Notation ............ 57
    5.3.2 The GrammarSketch Notation .......................... 59
    5.3.3 Using GrammarSketch to Match Method Declarations ... 61
  5.4 Extracting and Analyzing Pattern Matches for Dependencies 65
    5.4.1 Compiling the GrammarSketch Patterns ................ 66
    5.4.2 Applying the Patterns Against the Source Code ........ 67
    5.4.3 Conducting the Dependency Analysis .................... 68
  5.5 Using GrammarSketch to Identify and Trace Relationships ... 69
  5.6 Summary ..................................................... 73
6 Evaluation and Results ............................................ 76
  6.1 Goals of the Evaluation ..................................... 76
    6.1.1 Research Question 1 .................................... 77
    6.1.2 Research Question 2 .................................... 77
  6.2 Methodology ................................................ 78
    6.2.1 Evaluation Target: The OpenBravo ERP System .......... 78
    6.2.2 Participant Recruitment and Selection .................. 79
    6.2.3 Evaluation Overview .................................... 80
  6.3 Results ....................................................... 81
    6.3.1 Characterization of Dependencies ....................... 81
    6.3.2 Participant 1 (P1) ...................................... 83
    6.3.3 Lessons Learned From Participant 1 ..................... 84
List of Tables

4.1 Island Grammar Detection of Polylingual Dependencies. 45
6.1 GrammarSketch Detection of Polylingual Dependencies 95
8.1 Comparison of Luther and GrammarSketch configuration accuracies. 105
C.1 Improvements made between the lexical, keyword, and nesting grammars 118
C.2 Improvements made to the references grammar 119
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Conceptual overview of the JPetStore system.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>SQL, XML, and Java dependencies facilitated by iBatis in JPetStore.</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>JPetStore’s XML specification to add a new user’s credentials.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>HTML, JavaScript, and Java dependencies facilitated by Java Server Pages (JSP) in JPetStore.</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Excerpt from the NewAccountForm.jsp file.</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>JSP and Java dependencies facilitated by Session Beans in JPetStore.</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Code example from the VerticalMenu class in OpenBravo.</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of Luther.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>An island grammar production for locating method declarations.</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Excerpt from VerticalMenu.java.</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Excerpt from Menu_data.xsql.</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Screenshot of the GrammarSketch tool.</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Architecture of the GrammarSketch plugin.</td>
<td>55</td>
</tr>
<tr>
<td>5.3</td>
<td>Example code snippet.</td>
<td>62</td>
</tr>
<tr>
<td>5.4</td>
<td>Writing GrammarSketch Patterns in the /gsk/grammarsketch.gsk file.</td>
<td>70</td>
</tr>
<tr>
<td>5.5</td>
<td>Activating GrammarSketch to show matches for the patterns written.</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>GrammarSketch highlighting pattern matches.</td>
<td>72</td>
</tr>
<tr>
<td>5.7</td>
<td>Activating GrammarSketch to show all dependencies associated with an identifier.</td>
<td>73</td>
</tr>
<tr>
<td>5.8</td>
<td>List of identifiers dependent on the selected identifier.</td>
<td>74</td>
</tr>
<tr>
<td>5.9</td>
<td>“Drilling-Down” a dependency.</td>
<td>75</td>
</tr>
<tr>
<td>A.1</td>
<td>Certification of Institutional Ethics Review Letter</td>
<td>113</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

And the LORD said, “Indeed the people are one and they all have one language, and this is what they begin to do; now nothing that they propose to do will be withheld from them. Come, let Us go down and there confuse their language, that they may not understand one another’s speech.”

Genesis 11:6-7 (NKJV)

Software developers tasked with maintaining or evolving existing software systems spend considerable time exploring and understanding the source code of the system(s) that they intend to modify [SLVA97]. Over time, they expect that such software systems will undergo significant changes as new features are introduced, bugs are fixed, and support for new platforms (e.g., new operating system versions) are added. In many cases, modifications can be so severe and complex that the system may retain little resemblance to its original release. Documentation may not be useful to explain the current behaviour of the software as it is often out-of-date [LVD06, Sin98, BM97], or may have made the wrong assumptions as to what was important to document [LvV05, LVD06]. Even the knowledge held by the original developers of the system may not be relevant to its current behaviour [WSH01]. Consequently, the source code is often the first and sometimes the only resource a developer has to understand the underlying behaviour and architecture of a system [Sin98, LVD06]. Since understanding the interactions between various parts of the system is important for understanding the behaviour, the architecture, and the impact of any changes made to the source code, developers use a variety of techniques that involve detecting and tracing the dependency relationships in the source code of a system [RCM04, SMDV06, HW07].
Dependency analysis is laborious to undertake manually for a reasonably sized system [Boh02b], so programmers rely on tool support to reduce the effort involved in identifying and investigating dependency relationships. However, previous research has shown that dependency analysis is undecidable [Lan92, MW90]; tool support can only approximate the real dependencies present in the system. Further, it is increasingly the case that software systems are being created and maintained using multiple programming languages [KLW06, WSH01]. To understand the dependency relationships in such polylingual systems, the developer’s tool support needs to not only recognize the syntax of several different programming languages, but must also analyze the syntax to determine where paths of execution in the software can cross the boundaries of each programming language. Interoperability between multiple programming languages is typically facilitated by an external entity: a protocol (e.g., CORBA\(^1\), Web-services\(^2\)), a library (e.g., the Java Database Connectivity library\(^3\) for executing SQL queries from within Java), or an architecture (e.g., Microsoft’s .NET platform\(^4\)). As a result, developers either must create special-purpose tools for arbitrary language combinations—an expensive prospect [Big94] —or must resort to inaccurate but language-independent lexical tools.

There are several characteristics of the problem of polylingual dependency analysis that hint at less expensive ways of providing tool support for developers. First, while providing tool support for the general case may be prohibitively expensive, in practice we seem to find software systems with only a handful of programming languages present. Secondly, the amount of information that needs to be known about each language to provide useful, approximate dependency analysis support is much less than the complete set of syntax and semantics for the language [Moo01]. If there are only a few languages and technologies in use for a specific software system, and the level of semantic understanding needed to approximately detect

\(^1\)www.omg.org/technology/documents/formal/corba_iop.htm, as of 2008/08/05
\(^2\)www.w3.org/TR/soap, as of 2008/08/05
\(^3\)http://java.sun.com/javase/technologies/database, as of 2008/08/05
\(^4\)http://www.microsoft.com/NET, as of 2008/08/05
polylingual dependencies is low enough that a developer could reasonably be expected to define such semantics on their own, then it may be possible to provide a lightweight framework for generating tool support that can be easily configured by individual developers for their own work. The question I am exploring in this work is: what semantic properties in programming languages can be used effectively to approximately recognize the polylingual dependencies in a software system?

1.1 Background: Evolving Software Systems

From the day that it is released to its users, a software system begins to degrade [Par94]. This notion that software can decay is a strange one, considering that software is in some respects a collection of algorithms and mathematical formulas that should be immune to the ravages of time. Our notion of software decay has nothing to do with corruption of source code or the executable portions of a system; rather, a software system decays over time if it is not adapted to fit the changing needs of its user-base and environment [Par94].

1.1.1 The Role of Dependency Analysis

Enacting any change to a software system is difficult to do immaculately; the complex nest of relationships or dependencies present in software make it likely that any change in the behaviour of some component will have a “ripple effect” across the software system [PC90, Boh02b, Boh02a]: other portions of the system may cease to function correctly if the change in behaviour violates the assumptions and expectations that are relied on for operation [LvV05]. Identifying these potential consequences of modifications to software is referred to as change impact analysis [AB93]. Impact analysis techniques, such as the application of dependency analysis, are employed as part of investigating and planning such modifications to software. For example, a developer might use a dependency analysis tool configured with some level
of understanding of the semantics of the programming language in use to recover structural relationship information from the source code (e.g., caller–callee relationships, inheritance hierarchies, usage of common types or classes). Based on this information, the developer forms a picture of the consequences of the proposed modification and can alter their planning accordingly.

**Sifting Source Code for Relevant Facts**

Tracing dependency relationships in source code requires that a developer have some understanding of the semantics of the programming language in use, both to recognize where dependencies exist and to understand what portions of source code are less relevant for their investigation. Typically, the portion of a programming language’s syntax that is of interest to a developer for dependency analysis is a small portion of the complete language specification [Moo01]. A developer will sift through source code looking for *key structural details* in the code that suggest the presence of a dependency. For example, when examining a file written in the Java programming language, a developer conducting dependency analysis will likely be most interested in the class and method declarations, and references to types or method invocations elsewhere in the system. Statements in the code that describe the system’s control flow would be overlooked; such details could be useful, but by ignoring them the developer simplifies their investigation task.

Any analysis of the dependencies in a software system is difficult, and because of the complexity of such analysis it is almost impossible to know if one has discovered all of the dependencies present in the system [PC90, MW90, Lan92]. Developers are responsible for making *tradeoffs* between the accuracy of any dependency analysis they conduct, and the difficulty or cost of obtaining that accuracy. In some contexts, the developer may not need their analysis to be extremely accurate as they are simply investigating a change, or the cost of making a mistake in the modification to the software is quite small. In such cases, a developer may
look to understand just enough of the language semantics under analysis to obtain a reasonable approximation of the real dependencies in the system.

1.1.2 The Problem of Polylingual Software Systems

A multiple-language, or polylingual, approach to programming allows developers to leverage the advantages that a particular language brings to a solving a problem (e.g., SQL is frequently used for database manipulation), or because the developer wishes to use the libraries, APIs, and platforms available through a particular language. Communication between source code written in differing languages is usually facilitated by a third party library or technology: a Java program for example can utilize code written in C through the Java Native Interface (JNI)\(^5\), even though the Java compiler does not recognize source code written in C.

Such systems have dependency relationships crossing language boundaries due to these facilitating libraries and architectures, but existing language-specific tool support fails in this context because while it may have a sophisticated understanding of one language, the understanding of other languages and the mechanisms by which cross-boundary communication is undertaken is missing: the tool ceases to become useful once a developer needs to trace relationships across a language boundary. The alternative is for the developer to consider creating special-purpose versions of these tools for their needs—an expensive prospect—or utilize lexical tools like grep that are language-independent but are also inaccurate and require significant configuration work on the part of the developer.

1.2 Overview of Related Work

Existing techniques for supporting polylingual dependency analysis are generally either lexically- or semantically-based. Lexically-based support is flexible and inexpensive to implement but sacrifices any knowledge of the language under analysis, relying on only the developer's skill

\(^5\)http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html, as of 2008/08/05
in writing the appropriate pattern expressions [MN96, AG06]. Semantically-based approaches encode knowledge of the languages’ syntax and interaction protocols into the tool support, requiring less effort by the developer to use effectively [MW05, FF06, MX06]. This ease-of-application comes at a high cost: The nature of the interactions between program fragments written in different languages is rarely standardized, being typically facilitated through the use of protocols, libraries, and frameworks external to the languages in use. For example, SQL has been used to embed database queries as string literals inside Java or C++ source code [MX06], but a library is required to facilitate the execution of the SQL query. As a result, each combination of languages (and the particular interaction protocols in use) will require specialized support, leading to exponential growth as the number of cases increases [Big94].

1.3 The GrammarSketch Tool

Lexical tool support is perhaps the most attractive solution for polylingual dependency analysis, simply because it remains the most flexible for the language and technology combinations encountered. Lexical tools tend to be lightweight, requiring a reasonable amount of effort by the developer to configure the tool appropriately for their system [MN96]. Lightweight approaches are desirable because of the variability in the languages and technologies encountered; the effort investment needed to develop a more powerful tool may be unreasonable for the few systems it would be appropriate for [Moo01]. However, lexical approaches achieve this flexibility and ease of configuration by sacrificing the deeper details of a language it analyzes. Being experts on the systems that they maintain [Sin98], developers could likely describe some semantics of the languages or technologies they need to analyze, if such a specification was not onerous. What semantics then are worth recognizing?

My approach was to first conduct an exploratory study with a polylingual system. I evaluated several different language semantics across three programming languages used in the
system to understand how configuring tool support to recognize those semantics improved the resulting accuracy, but also compared that to how difficult configuring tool support was for those semantics (see Chapter 4). My work identified that types and references seemed to provide a significant improvement in the precision of dependencies detected, but were still relatively expensive to configure tool support to recognize.

Based on these results, I explored developing lightweight tool support for polylingual dependency analysis that incorporated two ideas:

1. presume which language semantics are the most to effective to recognize for polylingual dependency analysis between two arbitrary languages; and,

2. require that a developer specify just those syntactic patterns necessary to recognize such semantics, for each programming language used in their software system

GrammarSketch is a research prototype of such tool support. The prototype provides a simplified, regular expression based notation, that can be used by a developer to describe the syntax of the semantics that GrammarSketch has imposed on them as being important to recognize for polylingual dependency analysis. Using these developer-specified syntactic patterns, GrammarSketch analyzes the developer’s source code and infers the presence of dependencies in the source code by using a set of assumptions as the imposed semantics in each language interact across language boundaries to form dependencies.

1.4 Thesis Statement

The thesis of this work is that lightweight polylingual dependency analysis tool support can be provided by imposing on the developer which semantics must be recognized for dependency analysis, and using the developer-specified syntactic patterns for those semantics to generate approximate tool support comparable in effectiveness to lexical analysis alternatives.
1.5 Overview of Thesis

The rest of the thesis is organized as follows: Chapter 2 describes an example scenario outlining the difficulties inherent in trying to trace polylingual software dependencies, and how a developer would currently attempt to solve them. Chapter 3 outlines the work done by other researchers that is relevant to this problem. Chapter 4 describes my initial work on this problem using the Luther testbed to evaluate the *island grammars* approach to understand the tradeoffs between the accuracy of an island grammar and the cost to configure the grammar. Chapter 5 describes how the results and lessons learned from the Luther testbed were applied to the development of the GrammarSketch tool, which provides a new approach based on developers quickly outlining approximate patterns for detecting dependency relationships. Chapter 6 discusses my evaluation of the tool, while Chapter 7 concerns the implications of the results, and the direction of future research in this area. Chapter 8 concludes the thesis and explains its contributions.
Chapter 2

Motivation

The goal of this chapter is to illustrate for the reader why dependency analysis in a polylingual software system for even a conceptually simple change can be extraordinarily frustrating. In Section 2.1 I present an example of a change task a developer might wish to investigate on JPetStore, a polylingual software system. In Section 2.2, I walk the reader through what such an investigation would look like, and where the developer would encounter difficulty due to ineffective or absent tool support. In Sections 2.3 and 2.4, I present examples illustrating the difficulties inherent in using existing dependency analysis techniques: Section 2.3 provides an example of using regular expressions for lexical analysis, while Section 2.4 discusses the difficulty of creating or adapting syntactic analysis support.

2.1 Scenario: Modifying the JPetStore Web-Based System

Consider a developer who is creating an online web-store for their company’s employees to access over an internal network. The intent is to allow employees to browse and order company-branded clothing and merchandise. He decides to build off of JPetStore, a small-scale web-store system that ships as an example with the data mapping framework iBatis\(^1\). The developer knows that he will need to replace the login mechanism used by the JPetStore system with one that leverages the company’s existing network login credentials for each employee, as stored elsewhere in a database. He decides to base part of his feasibility study for re-using JPetStore on a rough estimate of the impact this change would have to the system.

\(^1\)http://ibatis.apache.org, as of 2008/08/05
2.1.1 JPetStore: An Architecture and Language Overview

JPetStore follows a classic three-layer architecture for a web-based system, illustrated in Figure 2.1. The layers, and the languages in use in each of these layers are as follows:

- Data Layer: This layer houses the facilities that interact with the system’s database, and typically handles all requests to retrieve or store information. In JPetStore, the source code for the Data layer comprises database manipulation queries written in SQL and stored in element tags in XML files.

- Logic Layer: This layer is responsible for the behaviour of the entire system. It responds to client requests, understands what information it needs to retrieve from the Data layer, and what information needs to be delivered to the Presentation layer. In JPetStore, the Logic layer is exclusively written in Java.

- Presentation Layer: This layer is what the end-user sees in their web-browser: a series of webpages that display what the store has for sale, and lets the user make the appropriate selections. The Presentation layer is responsible for sending user input to the Logic layer, and for changing the webpage contents based on the feedback from the Logic layer. The Presentation layer in JPetStore is comprised of several different kinds of source code:
most of the webpages are principally composed of HTML tags mixed with plain text, but may also embed JavaScript\(^2\) or Java code in the webpages.

All three of these architectural layers are written using incompatible programming languages\(^3\), in the sense that the languages have no specification as to how they may operate with these other languages. Communication inside, and across, these architectural layers is facilitated by several different protocols, libraries, and frameworks. Rather than bog the reader down in details as to how polylingual communication is facilitated, we will instead address the various technologies as they crop up during the dependency analysis.

### 2.2 Conducting a Dependency Analysis on JPetStore

![Diagram of dependencies facilitated by iBatis in JPetStore.](image)

**Figure 2.2:** SQL, XML, and Java dependencies facilitated by iBatis in JPetStore.

JPetStore uses a series of XML files to define an object-relationship mapping between Java classes and SQL queries (illustrated in Figure 2.2), for which the iBatis framework can then

\(^2\)Despite the name, JavaScript has almost nothing in common with the Java programming language. Officially it is referred to as ECMAScript, but colloquially is still called JavaScript for historical reasons [ECM99].

\(^3\)XML, HTML are not programming languages - they would be more accurately referred to as data definition or markup languages. However, this distinction is not important for our purposes as a failure to properly update XML or HTML code on which the system is dependent will still cause a defect. They will be collectively referred to as programming languages for the sake of simplicity.
generate Java objects at run-time. Using a text editor, the developer looks for the `sql-map-config.xml` file, that defines the XML mappings in use, and decides that `Account.xml` may be the most relevant. Browsing the queries in the file, he notices that account information is stored in the `SIGNON` table, which has two columns: `USERNAME` and `PASSWORD`. One of the relevant SQL queries for adding a new user’s credentials (from `Account.xml`) is presented in Figure 2.3.

```xml
<sqlMap namespace="Account">
  <typeAlias alias="account"
    type="com.ibatis.jpetstore.domain.Account"/>

  <insert id="insertSignon"
    parameterClass="account">
    INSERT INTO SIGNON (PASSWORD,USERNAME)
    VALUES (#password#, #username#)
  </insert>
</sqlMap>
```

Figure 2.3: JPetStore’s XML specification to add a new user’s credentials.

As a first attempt, the developer runs a case-insensitive lexical search on the entire system to see what other files also contain the `USERNAME` identifier; the search returns 106 matches; in looking through the results, he recognizes that most represent false positives. He restricts the search to just the `Account.xml` file: three other queries that access the `USERNAME` column are found. He is reasonably confident that these are the only three queries that he has to worry about, as the names of the other XML files do not suggest that they deal with user authentication. His next step is to identify what Java functionality is dependent on these queries. To do this, he must understand the semantics of the iBatis framework in mapping a query to a Java method. As an example, in Figure 2.3 the value of the `id` attribute in the `insert` node is the same as the first parameter in `update("insertSignon", account)`, a method invocation inside the `insertAccount()` method in the `AccountSqlMapDao` class. The class extends
part of the iBatis framework, and uses the update method it inherits to perform the query lookup at run-time. He manually determines that a total of four methods in the `AccountSqlMapDao` class are dependent on the authentication queries (two overloaded `getAccount(...)` methods, and `insertAccount(...)` are the other three).

To continue his feasibility study, the developer determines what other parts of the system are dependent on these four methods in the `AccountSqlMapDao` class. After building the JPetStore system in an Integrated Development Environment (IDE) like Eclipse, he can use the semantic tools it provides to quickly infer what classes are dependent on these methods. He determines that three classes are dependent on the four `AccountSqlMapDao` methods: `Account`, `AccountBean`, and `AccountService`.

2.2.1 The Relationship Between the Java and the Java Server Pages (JSP) SubSystems

Figure 2.4: HTML, JavaScript, and Java dependencies facilitated by Java Server Pages (JSP) in JPetStore.

The developer now has a problem: he knows that several webpages (forming the user interface for this system) accept login information, and likely interact with the Java code. But unlike the XML files, there is no contextual information in the Java code to suggest which pages

\footnote{http://www.eclipse.org, as of 2008/08/05}
those are, or what entities in the Java code the functionality is dependent on. The developer
resignedly sets up a Tomcat server on his machine, deploys the JPetStore system, and man-
ually investigates the webpages. He guesses that NewAccountForm.jsp, EditAccount-
Form.jsp, and SignOn.jsp are the most relevant. These webpages are written using Java
Server Pages (JSP) (illustrated in Figure 2.4), and make use of yet another framework called
Apache Struts to dynamically generate content.

```
<table>
  <tr><td>User ID:</td>
   <td><html:text name="accountBean"
           property="username"/>
  </td></tr>
  <tr><td>New password:</td>
   <td><html:password name="accountBean"
           property="password"/>
   </td></tr>
</table>
```

Figure 2.5: Excerpt from the NewAccountForm.jsp file.

![Diagram showing JSP and Java dependencies facilitated by Session Beans in JPetStore.]

Figure 2.6: JSP and Java dependencies facilitated by Session Beans in JPetStore.

Getting annoyed at the time this “estimate” is taking, the developer opens the three web-
pages in his IDE. Again, an understanding of how the Struts framework encodes dependency
information is needed for him to manually and precisely determine what dependencies exist between each JSP page and the Java source code (illustrated in Figure 2.6). In Figure 2.5, the developer knows that the name attribute refers to a Java object, and property refers to a field on the same object. Using a case-insensitive lexical search, he may correctly infer that "accountBean" is referring to the AccountBean class, that he identified earlier as depending on the authentication queries. But a lexical search on the fields within AccountBean will not yield a match for username or password. The developer must remember that the Struts framework allows aliases to be defined in a separate file that might not be in the same directory as the rest of the source code. After some blind poking around in the project directories, he finds the message.properties file that indicates that username and password are aliases for fields in the Account class, an instance of which is also stored as a field in the AccountBean class. The developer must now repeat this manual analysis on the other two webpages before coming up with an estimate of the change impact on the JPetStore system in replacing the login mechanism.

2.2.2 How Can This Go So Wrong?

What the developer wanted was to select a few queries he thought would change, and quickly see a set of classes and webpages that could be affected. Since it was an exploratory investigation, some inaccuracy would have been acceptable. Instead, he ended up trudging through a largely manual, method-level investigation of the code with tool changes, context switches, and even a system deployment, being obstacles to getting his answer. While JPetStore is a small-scale polylingual software system, even conceptually-small change tasks can require demanding dependency investigation across multiple programming languages, protocols, and technology platforms.
public void doPost (HttpServletRequest request,
            HttpServletResponse response)
        throws IOException, ServletException {

            VariablesSecureApp vars =
            new VariablesSecureApp(request);

            if (vars.commandIn("DEFAULT")) {
                printPageDataSheet(response, vars, "0", false);
            } else if (vars.commandIn("ALL")) {
                printPageDataSheet(response, vars, "0", true);
            } else {
                throw new ServletException();
            }

        }

Figure 2.7: Code example from the VerticalMenu class in OpenBravo.

2.3 Regular Expression Based Lexical Analysis

The motivational example presented in Section 2.2 illustrates some of the conceptual difficulties involved in conducting polylingual dependency analysis on a system. It obscures the difficulties present in using some of the tools, particularly on how difficult it can be to configure the lexical analysis tools that are commonly used in polylingual dependency analysis. In this section, I present an example of how to write a Perl-style\(^5\) regular expression to detect a method declaration in Java. In doing so, I intend to show both how much effort an individual developer is expected to expend to configure such tool support on their own, but also illustrate how unwieldy the end-product is.
2.3.1 Example: Defining a Basic Identifier Pattern

Figure 2.7 is a snippet of code from the org.openbravo.erpCommon.utility.VerticalMenu class in the OpenBravo ERP system, that illustrates the syntax of method declarations in Java. To write a regular expression to capture all such method declarations, one might begin by trying to write a pattern to at least recognize the name of the method. We start by using the w pattern, which accepts a “word” character (any letter, number, or underscore), and use the + modifier, which indicates that this should be a sequence of one or more such characters:

\w+

But this is not quite correct: the Java programming language requires all identifiers to begin with a letter or an underscore. To fix this, we modify the pattern to ensure that the starting character is only a letter or an underscore:

[a-zA-Z_]\w*

The * modifier indicates that we can match a word character zero or more times. So this will match an identifier of length one to infinity. But this will match all such identifiers in the system; we only care about those that describe method declarations.

2.3.2 Example: Expanding the Pattern to Find the Method Name and Return Type

In Figure 2.7 two reserved words precede the method’s name: the public keyword describes the method’s visibility, and the void keyword indicates the return type of the method. We might try repeating the earlier pattern we wrote for identifiers three times, and separate each pattern with the \s+ pattern (which matches any single whitespace character one or more times): \n
5 The syntax presented in this section is used in the Perl programming language, and has been adopted by several other languages (like Java) and technologies.
6 (Spaces have been added for clarity)
This pattern though will not match a sizable number of method declarations for several reasons:

- Java does not require the visibility modifier to be present. If a method is not declared to be public, protected, or private, it is assumed to be protected.

- There are several other modifiers that might have been added to the method’s declaration. It may be declared (as of Java 1.6) static, final, synchronized, native, volatile, and strictfp. So the number of identifiers in front of the method’s name can arbitrarily range from one to eight.

Rather than deal with this, we change the approach: capture two consecutive identifiers, assume the first identifier is the return type, and the second identifier is the method’s name:

\[
([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+}
\]

2.3.3 Example: Adding Recognition of Parameter Lists

Any method declaration has, after the method’s name, a series of parameters enclosed in side a set of parentheses. The possibilities are:

- No parameters declared.

- One parameter declared.

- A list of parameters declared, each pair separated by a comma.

So we need to write a pattern that can handle all three of these cases at once:

\[
([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+} \ ([a-zA-Z_]+) \ \text{s+}
\]
Already, the regular expression pattern is getting quite ugly and unreadable, yet is still not
done, and now is also wrong. While there must be at least one space separating identifiers in
Java (otherwise, how does one know the identifiers are distinct?), there is no such restriction on
spaces between identifiers and punctuation (e.g., , and ). Furthermore, as written the pattern
implies that if there is no parameter list there must be still at least two spaces between the
parentheses. So we need to modify the above pattern to read as follows:

```
([a-zA-Z_]\w*) \s+ ([a-zA-Z_]\w*) \s* " \s* (([a-zA-Z_]\w*)
\s+ ([a-zA-Z_]\w*))? (\s* "," \s+ ([a-zA-Z_]\w*) \s+
([a-zA-Z_]\w*))\s* \s* ")
```

2.3.4 Example: Dealing With Exceptions in the Method Declaration

To increase the precision of the pattern, we add the recognition of the first brace in the method
declaration to the pattern. However, between the end of the parameters list and the start of
the method’s body we may encounter a throws clause, and a series of exception types. The
exception types may or may not be important to us, but they will need to be addressed. If we
wish to ignore them, we can use the pattern to match any character up to the beginning of
the brace:

```
([a-zA-Z_]\w*) \s+ ([a-zA-Z_]\w*) \s* " \s* (([a-zA-Z_]\w*)
\s+ ([a-zA-Z_]\w*))? (\s* "," \s* ([a-zA-Z_]\w*) \s+
([a-zA-Z_]\w*))\s* \s* ") .+ "{"  
```

If the exception types are, in fact, important to capture:

```
([a-zA-Z_]\w*) \s+ ([a-zA-Z_]\w*) \s* " \s* (([a-zA-Z_]\w*)
\s+ ([a-zA-Z_]\w*))? (\s* "," \s* ([a-zA-Z_]\w*) \s+
([a-zA-Z_]\w*))\s* \s* ") \s* "throws"? ([a-zA-Z_]\w*)\s* 
\s* "," \s* ([a-zA-Z_]\w*) \s* "}
```

```
The use of the `?` operator indicates that the pattern preceding it (e.g., "throws") may or may not be present. While we are interested in capturing the type information of the exceptions thrown, Java does not require that every method have a `throws` clause; our pattern must accommodate both cases.

2.3.5 Example: Limitations of the Pattern

The regular expression pattern constructed above will likely match nearly all of the method declarations in Java code we are interested in, but the pattern is far from perfect. There are several flaws with the pattern:

- The pattern assumes the Java source code it is applied against is syntactically correct; there are many possible input strings this pattern will accept and match that are incorrect Java code.

- The pattern still ignores other syntax cases that may occur in the method declaration. As written, the pattern fails to handle:
  - array declarations (e.g., `String[]`),
  - generics (e.g., `List<String>`),
  - declaring method parameters as `final`,

and likely numerous other syntax patterns.

Addressing these additional conditions would further complicate the pattern description, which is already approaching illegibility due to the cases it already attempts to deal with.

2.3.6 Summary of Example

The above represents the final form of the regular expression pattern necessary to recognize method declarations, but even to a developer trained in writing such patterns it must appear
almost nonsensical. If the programming language’s syntax changed over time (e.g., the addition of support for Generic types in Java), the above pattern would be difficult to change, and highly susceptible to being broken due to a single, difficult to detect mistake. Further, this pattern will only match method declarations; this forms only part of information that a developer would be interested in extracting for dependency analysis. Several more such patterns would need to be written to capture all those key structural details in every programming language that a developer wished to use such lexical dependency analysis techniques on.

2.4 Grammar Based Syntactic Analysis

The work necessary in Section 2.3 to detect method declarations in Java is unnecessary if one is content use more powerful syntactic analysis tools; the Eclipse IDE has powerful syntactic analysis facilities for Java that already facilitate such searches, and even provides a variety of means by which the developer can explore the results. Such techniques are heavyweight, requiring a significant investment of effort to create, but result in far more effective tool support that often requires little effort on the part of another developer to use.

It is difficult to present an example similar to that of Section 2.3 to show how difficult it is provide syntactic analysis support for the same problem, because syntactic tools are instrumented to analyze a complete language syntax, rather than just the syntax for a specific language semantic of interest [Gri06]. Developing such support for an entire language can be an expensive prospect.

- The Ruby Development Toolkit (RDT)\textsuperscript{7} is a plugin for the Eclipse IDE that adds syntactic analysis support to Eclipse for the Ruby programming language, similar to what is already provided for Java. To provide this support, the developers of RDT needed to create 519 Java classes, comprising 62,399 lines-of-code.

\textsuperscript{7}http://rubyeclipse.sourceforge.net, as of 2008/08/05
• The complete Java language specification is currently 684 pages in length. This contrasts with only 553 pages for the C# language specification.

• One researcher has reported the development of a COBOL language grammar requiring 4 months of effort [Moo02].

The high cost of developing syntactically-aware tool support usually restricts the development of such support to large teams or organizations, where the effort required to develop such support can be amortized over a large user base, or offer enough savings in developer time and defect rate to make the investment worthwhile.

In cases where syntactic analysis techniques are used, practitioners often rely on some other party to provide the syntax analysis support they need before applying their techniques (see [AG06, MW05, FF06, MX06]). But this is not always a practicable solution; in many cases, the format in which syntactic analysis support is provided is tailored for the convenience of the tool it was created for, and is not easily adapted to other uses [AG06]. If the language the syntactic analysis support is developed for has changed over time, significant modifications may be required; the reference grammar for the JavaCC parser generator for the Java programming language doubled in size between versions 1.02 and 1.5 of the language. More significantly, such syntactic support has proven brittle when adapted to new languages or new forms of analysis [Gri06, MN96].

2.5 Summary

The current state-of-the-practice involves developers tracing relationships through multiple programming languages, using multiple, mutually incompatible tools for the analysis that put
most (if not all) of the responsibility for correctly understanding the language and communication semantics squarely on the developer. Conceptually straightforward modifications to polylingual software systems are encumbered by investigations that require combinations of tool support. Existing tool support ranges from being too difficult to configure properly, too restrictive to be generally useful, or too inaccurate to be useful. Even lightweight, lexical analysis approaches like regular expressions can become incomprehensible as they attempt to deal with the variations of syntax found in source code. Heavyweight techniques are so expensive to create, or difficult to use, that they tend not to be a realistic option unless someone else is willing to incur the expense of creating them.
Chapter 3

Related Work

In this chapter I discuss other research that has been directed at solving the specific problem of polylingual dependency analysis, or has particular relevance and usefulness for this problem. Section 3.1 details some of the contexts in which polylingual software development is occurring in industry, and Section 3.2 discusses early research addressing dependency analysis. Sections 3.3, 3.4, and 3.5 outline research related to polylingual dependency using lexical, syntactic, and hybrid approaches respectively.

3.1 Introduction

The JPetStore example in Chapter 2 is not a special case; a polylingual approach to software development is increasingly commonplace. Embedding SQL query fragments inside code written in another language (e.g., Java or C++) to communicate with databases has been a programming practice for some time (e.g., see [BKM90]). The J2EE 1.4 Platform used XML to supply additional dependency information for classes, including embedding database queries written in an SQL dialect within XML, then mapping the queries to methods in a class [PCS03]. Some protocols are independent of the implementing languages, such as web services¹ and the Common Object Request Broker Architecture². Others target specific language combinations, like the Java Native Interface which bridges Java and C/C++ code [FF06, MW05]. The semantics describing the cross-language dependencies in each system are often specific to the technology used, and not generalizable.

¹www.w3.org/TR/soap, as of 2008/08/05
²www.omg.org/technology/documents/formal/corba_iop.htm, as of 2008/08/05
3.2 Dependency and Impact Analysis

Early work in dependency analysis took a highly formal approach, but was quick to recognize the need for approximation (even with precise definitions of language semantics) and the important effects that subtly different definitions of dependencies could have [PC90, MW90, Lan92]. More pragmatic approaches to impact analysis followed [AB93], but still made strong assumptions about having the intended change implemented in order to perform the analysis.

3.3 Lexical Approaches to Dependency Analysis

Lexically-based tools require the developer to define regular expression patterns to recognize dependency structures in code. Developers can then configure such tool support for the languages or technologies that are peculiar to the software systems they wish to analyze. The most well-known lexical analysis tool is the grep [Bou77] family of tools, which provide support for matching regular expressions on any text source.

3.3.1 The Lexical Source Model Extraction (LSME) Tool

A more refined example is the Lexical Source Model Extraction tool [MN96] developed by Murphy and Notkin. Developers define lexical patterns to match relevant source code using regular expressions, and also provide a second set of specifications to instruct LMSE on what to do when a lexical pattern finds a match in the source code. The instructions allow developers to combine matched patterns to form a model of their own choosing for the source code. The LSME tool improves on several of grep’s limitations:

- LSME provides implicit tokenization of the source code: the tool clusters together any sequence of characters that are not separated by whitespace or special “escaped characters”, and treats them as a single word. The developer can write lexical patterns to
operate on these words, sparing developers from having to specify what constitutes an identifier for every language analyzed.

- The tool allows patterns to share a hierarchical relationship, such that a pattern may have a requirement that another pattern must first be matched in the source code before it may match anything. This allows developers some flexibility in constructing fine-grained patterns by combining smaller patterns together.

- LSME provides a separate mechanism for operating on matches found in the source code, allowing developers a means of combining lexical patterns in more sophisticated ways than `grep` allows.

LSME has several significant limitations in the context of polylingual dependency analysis. The tool operates at a lexical level of analysis, so it cannot leverage even basic knowledge about the syntax of languages under analysis. For example, LSME has no understanding of which identifiers in a programming language are reserved, and so special checks must be written to detect if a word that LSME classifies as an identifier is actually a reserved word and should be ignored. Secondly, the tool is intended for a broader range of applications than just dependency analysis; one of the concerns in their work was ensuring that the notation language for LSME had the same expressiveness as the regular expression language it is based on. Consequently more information is needed to describe those patterns necessary for dependency analysis than would be required if the tool was specifically targeted at dependency analysis. Finally, the tool may require more skill to use than the `grep` tools it is intended to replace as the developer is not only required to write patterns to configure the tool, but also instructions on what to do with any matches it finds.
3.3.2 The TAWK Tool

Atkinson and Griswold [AG06] attempt a different variant on lexical analysis with their TAWK tool, which operates on abstract syntax trees (AST) instead of plain text. Similar to the LSME approach, developers provide two specifications: one describes a specification for creating an AST representation for any source file written in a single programming language. The second specification applies lexical patterns to the generated AST to extract a model of the source code; regular expression patterns are written to match the text contained in the AST, but are qualified with additional restrictions on the categories of syntax nodes that can be matched. By allowing the developer to leverage a limited amount of syntactic information in writing lexical patterns for source code, the TAWK tool allows for greater precision than a traditional lexical analysis tool could provide. However, the TAWK tool relies on a toolkit (called Ponder) to generate the AST for any source code file it parses; at the time of publication only two languages were supported. TAWK than falls victim to the same problems as syntactic approaches (discussed in Section 3.4) where developers may be required to define a grammar for each and every language in a system they wish to analyze before they can use the tool. Further, generating an AST representation for a source code file requires a complete understanding of the syntax of the language that is more than would be minimally necessary to extract the key structural facts necessary for detecting polylingual dependencies.

3.4 Syntactic Approaches to Polylingual Dependency Analysis

Syntactically-based tools for polylingual systems are typically focused on the technologies that bridge languages for interoperability, and are configured for specific language combinations. Each of the research approaches in this section depend on having available tool support that recognizes the complete syntax of the programming languages under analysis, as well as some understanding of each language’s semantics as well.
All these approaches share some basic common strengths and weaknesses. Because the tools recognize the syntax and incorporate knowledge about some of the semantics of the languages they analyze, they are more likely to be accurate in their detection of dependencies because they are not confused by cases where there exists lexical similarity between two identifiers, yet their semantic context indicates they are not dependent on each other. However, providing complete syntactic or semantic recognition of the source code is an expensive prospect; in most cases, someone must construct a grammar describing the syntax of a programming language in a format proprietary to whatever tool is in use (compare [AG06], [MW05], [Moo01], and [CW07]: in each case, the format used for describing a grammar is distinct and incompatible) which is a non-trivial task, even for a researcher [Moo01]. If such support is not available, or is not maintained as revisions/dialects to a programming language are encountered, the onus falls on to the developer to enact such updates or to abandon the tool.

3.4.1 Research on Interactions between Java and C Using the JNI Technology

Moise and Wong [MW05] tried an approach conceptually similar to the TAWK tool described in Section 3.3 to find dependencies between Java and C code that used the JNI technology to facilitate dependencies. They used the syntax analysis support present in the Source Navigator IDE\(^3\) to extract key structural facts from the source code in each language, and then applied a series of heuristics to determine where dependencies between the two languages existed. A key result from their work was that the contextual information needed to infer a dependency between program fragments in different languages was not sufficiently provided by the Source Navigator tool, even though it had complete syntactic understanding of the languages involved.

Moise and Wong’s approach has several key limitations:

- The Source Navigator IDE has not been updated since 2003, and supports only a few

\(^3\)http://sourcenav.sourceforge.net, as of 2008/08/05
languages. At the time of the publication of their research, the IDE had not been maintained for two years and at least one of the languages (Java) supported by the IDE had revisions and updates made to the language (Java 2 Standard Edition 5.0). Their approach is tied to a technology platform that at the time of publication was likely already obsolete.

- Moise and Wong do not qualify the effectiveness of their approach; it is not clear what kinds of JNI dependencies their approach is more effective for, and where it has difficulty. They report an evaluation of the dependencies present in two systems written using both Java and C:

1. the Win32RegKey application, for which they provide an example of only one detected dependency; and

2. the Java-GNOME library for which they present the number of dependencies found from Java code to C code (over 3,000) and from C code to Java (over 2,000). There is no indication as to whether a portion of these were investigated to verify the accuracy of the detection. In the case of dependencies originating from C code to Java code, the authors are careful to distinguish them as “possible connections” as the context of the C dependencies is difficult to extract using the existing tool support.

Moise and Wong may also have overlooked an important result in their work by blaming a problem on the programmers who wrote the Java-GNOME library they analyzed. The authors analyzed 109 cases where their analysis found a point in the Java code from which a dependency should be traceable to some point in the C code, but for which a match could not be found. After investigation, the authors concluded that the mismatches were largely due to im-

---

4 http://sourcenav.sourceforge.net/online-docs/progref/index_pr.html, as of 2008/08/05
5 http://www.jcp.org/en/jsr/detail?id=176, as of 2008/08/05
implementations that did not correctly follow the JNI specification, but still functioned correctly. One set of such problems was due to portions of C code that interacted with Java code: the C code used a standard C-type int, instead of a special type jint imposed by the JNI technology for integers being passed to and from Java code. The authors blame the programmer’s “inexperience with the project” [MW05] for these errors.

However, the JNI compiler maps the jint type to C’s int type using a typedef reserved word. During compilation all occurrences of jint are replaced with int. While it may be a bad practice to use int instead of jint, it is also possible that the developer knows what the JNI technology is doing and is taking advantage of the equivalence of these types. By not considering this, Moise and Wong overlook the possibility that the documented semantics of a polylingual communication technology may differ from the actual implementations in source code, without a failure in the technology. This would suggest that precise and complete polylingual dependency analysis is more difficult than we might appreciate; a tool that is configured with the correct, documented semantics of a technology like JNI will still miss those polylingual dependencies that do not share the proper syntax or semantics in the source code, but after compilation are indistinguishable from a correct form.

Furr and Foster [FF06] take a different approach to dependency analysis between Java and C code that uses the JNI technology. They restrict their work to dealing with a specific problem in dependency analysis: the JNI technology strips away most type information from C code that interacts with Java code. This makes checking the type correctness of these interactions difficult. Their approach analyzes the semantics of the interactions of C and Java code information to infer the correct type information in the C code, and then embeds this missing information into the code in the form of additional parameters and/or specially defined types. This additional information is used to check type safety in the polylingual dependencies. Furr and Foster’s work explicitly highlights how the semantic context necessary for correct deter-
mination of polylingual dependencies may be missing from source code.

3.4.2 Research on Interactions Between Java and SQL Using the JDBC Library

Martin and Xie [MX06] expanded on the work of other researchers in dealing with interactions between Java code and databases through SQL queries. They reuse several existing research tools to:

1. Analyze the compiled Java byte-code to determine at what points calls are made to the Java Database Connectivity (JDBC) library, which they refer to as *hotspots*.

2. At each hotspot, they use a research tool to generate the set of all possible SQL queries that could be sent to the database at that point. Based on this set of potential queries, they attempt to cluster together Java methods that may manipulate the same sections of the database as sharing the same dependencies.

Their work addresses two important issues in conducting dependency analysis: determining precise points at which a dependency can occur, in this case facilitated by calls to the JDBC library, and conducting dependency analysis where one programming language is embedded inside another language. Martin and Xie’s publication was abbreviated in length, so some of the shortcomings of their technique may not have been addressable in that venue:

- Their work does not adequately explain how they are able to reconstruct the set of queries that may be submitted at a particular point to the JDBC library: SQL queries are often stored as embedded string literals in source code (which they clearly address), but may also appear as a set of such strings which are combined in some fashion during execution, or retrieved from external sources during program execution [HOM06]. It is not clear to what extent data-flow analysis is used in such cases, and what the tool does when an SQL query cannot be recovered from this analysis due to incomplete information.
• The authors have not indicated if there are any limitations as to the syntax for SQL that they support; most major database vendors have unique dialects of SQL that allow operations not permitted by other vendors.

• The examples discussed in the work seem to be for systems of trivial complexity. The “Classifieds” system\(^7\) is the second largest of the four presented, but is comprised of only 19 webpages. For such a system, the automatic generation of possible queries may be feasible, but it is not clear if this approach can scale to industrial sized systems.

3.5 Island Grammars: A Hybrid Approach to Dependency Analysis

Moonen [Moo01, Moo02] aims to combine the benefits of lexical and syntactic approaches to dependency detection by the application of island grammars. An island grammar comprises (1) a set of grammar productions (the “islands”) written to recognize a portion of a language’s constructs that are of interest, in this case for detecting structural dependencies, and (2) a set of productions that capture and ignore a broad range of input to elide those constructs that are not of interest (the “water”) [Moo01]. In doing so, a developer is able to provide some syntactic and semantic information about a programming language they wish to analyze, but is able to specify considerably less information about the language syntax than would be necessary to completely support the language. Moonen’s initial work was the Mangrove tool [Moo01], which used an approach similar to LSME: a notation format called syntax definition formalism or SDF is used to describe the island grammars for the language under analysis, than the developer can use either the Java, or ASF languages to describe what actions need to be taken upon finding a match.

Synytskyy et al. [SCD03] also applied island grammars in a proof of concept demonstration of fault-tolerant polylingual parsing. Their work involved developing and applying separate

\(^7\)http://www.gotocode.com/apps.asp?app_id=5, as of 2008/08/05
island grammars to parse HTML, Visual Basic, and JavaScript code present in a single, simple example file. While their work demonstrates such an application is possible, it is limited only to demonstrating the feasibility of such an approach on a trivial example. The authors do not use the island grammars to extract a model from the source code, or to conduct any dependency analysis. Consequently, their work offers little guidance as to how to proceed in such research, aside from suggesting that it is possible.

3.6 Summary

Researchers have explored varying paths for solving the problem of providing effective tool support for polylingual dependency analysis. Three main categories of research on this issue emerge:

1. lexical analysis techniques, based in some way on regular expressions, that are flexible enough to be adapted to this problem, but still require significant skill and investment on the part of their user to adapt to this problem;

2. syntactic analysis techniques, that are powerful and precise, but so expensive to create that researchers rely on the existence of pre-existing tool support that they can then adapt to their purposes; and,

3. hybrid analysis techniques, such as island grammars, that allow tool support to be syntactically aware for just those semantics of particular significance for dependency analysis, while ignoring syntax that is less useful.

Of the approaches discussed, the island grammars technique holds the most interest to me as a potentially useful way of achieving better-than-lexical precision in dependency analysis, while being cheaper to create than syntactic approaches.
Chapter 4

The Luther Testbed

This chapter presents the Luther testbed which was created to investigate the appropriateness of island grammars for developing tool support for polylingual dependency analysis. Luther was configured with a series of island grammars with varying levels of semantic richness, used to determine the dependencies present in set of polylingual source code. The dependencies extracted were compared with those manually determined ahead of time, and the precision and recall of that grammar’s accuracy was contrasted with the effort cost necessary to create and configure the grammar. The rest of this chapter is organized as follows: Section 4.1 provides a high level overview of the goals Luther aimed to address, and the design of the testbed tool. Section 4.2 describes the steps necessary to configure Luther for a particular set of programming languages, and with a specific level of semantic awareness. Section 4.3 explains the evaluation methodology used, and Section 4.4 presents the results.

4.1 Overview

In our motivational scenario in Section 2.2, the developer was looking for specific cues in sections of source code that indicated on what other functionality in the system it was dependent on. It was not necessary for him to read all of the source code to understand the system; it seems the syntax important for dependency detection is significantly less than a language’s complete specification. It may be possible to write tools for polylingual dependency analysis that approximate full semantic dependency detection by writing island grammars. However,

\footnote{Some of the contents of this chapter have been previously published by myself and Dr. Robert J. Walker \cite{CW07, Sections 4–7}. The formatting of these contents have been modified to integrate with the remainder of this thesis. \copyright 2007 IEEE. Reprinted, with permission, from Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM07). For the division of labour on the paper, please see Appendix B.}
defining what information must be recognized, and what can be elided is currently unclear. Our
goal is to understand when writing richer island grammars is economical. We have developed
Luther, a testbed tool designed to help evaluate the cost-to-accuracy tradeoff for the use of
island grammars in polylingual dependency analysis.

4.1.1 Architectural Overview

Figure 4.1 shows an overview of the architecture of Luther. Luther is configured with three
key kinds of information: (1) token attributes, (2) dependency rules, and (3) island grammars.
These are described respectively in Sections 4.2.1–4.2.3. Luther uses token attributes to de-
scribe the semantics of detected tokens as described in Section 4.2.1. Section 4.2.2 describes
how dependency relationships are used to match token attributes and infer semantic dependen-
cies between tokens. Island grammars, described in Section 4.2.3, are used to recover tokens
and their attributes from a language’s syntax.

Once configured, Luther accepts source code files as input, and activates the appropriate parser to extract tokens and their attribute information into a symbol table. When all files are parsed, Luther executes the dependency analyzer, which uses the data in the symbol table to generate a list of estimated dependencies. It outputs each pair of token dependencies to the console with their file-, line-, and column-locations in the source code indicated.

4.2 Configuring the Luther Testbed

4.2.1 Defining Token Attributes

A common set of attributes for tokens in all languages under consideration is provided for Luther to represent the semantic information present in the source code where that token occurs. The attributes so defined are encoded into a Symbol subclass that is responsible for storing the token and its attributes as part of Luther’s symbol table. These token attributes will be used to infer dependencies between tokens based on the semantics recovered by the island grammars.

Consider the following Java code snippet from the AccountSqlMapDao class in the JPet-Store: public Account getAccount(String username, String password) {...}. The token for getAccount has attributes to pinpoint the token’s location: AccountSql-MapDao.java (file), 24 (line number), 18 (column number). Since we are dealing with multiple languages, a language attribute (Java) is specified. An entity attribute indicates the semantic context of the token is a “method declaration”. We store what the token belongs to (the method is part of the AccountSqlMapDao class) in the parent attribute, and what belongs to the token (the username and password parameters, the contents of its statement block) in its children attribute.\(^2\)

\(^2\) Additional examples for Sections 4.2.1 and 4.2.2 are available in Sections C.1 and C.2.
4.2.2 Defining Dependency Rules

A set of dependency rules between attributes of two or more tokens are established for each type of dependency that Luther is to detect (e.g., between a method invocation and a method declaration). Each rule is encoded in the `DependencyAnalysis` class. Luther will then invoke this class against the symbol table after parsing is complete to determine what dependencies exist in the source code.

Continuing with our example in Section 4.2.1, line-, column-, and file-location attributes are clearly of little use in finding dependencies on `getAccount(..)`. Minimally, we expect that the dependency on `getAccount(..)` exists if and only if another token has the same name. The entity attribute of `getAccount(..)` (i.e., method declaration) restricts what other entities it can match (i.e., method invocations). The parent attribute points to its declaring class, whose name indicates a type. The parent of the method invocation token should be the token for the object it was invoked on, and that token’s attributes can be used to match it with its original declaration and hence type. Currently, Luther expects that all dependency rules comprising a dependency definition need to be satisfied for there to be a dependency between two tokens.

4.2.3 Defining Island Grammars

To create a parser for each language we wanted to analyze, we used the JavaCC 4.0 parser generator to compile grammar specifications into executable Java code. Grammar productions in the specifications were described using the island grammars technique for each language, as described in Section 3.5.

The Luther testbed provides a standardized template for each grammar to be built on that provides token definitions for C-style identifiers. The template also ensures that a compiled parser includes meta-data to indicate to Luther the language it is capable of handling, and provides a static method `lex` that serves as the entry point into the parser. `lex` accepts a `File`
reference to the code to be parsed, and a reference to a symbol table maintained by Luther. When invoked, it instantiates and executes the parser, storing collected data in the symbol table.

**Supporting embedded languages**

Source code from one language may be embedded in source code for another. The mechanism for delineating the embedded code varies across languages and technologies. In cases where the embedded code is clearly marked (e.g., in Figure 2.3, the contents of an XML element named `insert` are a SQL query), JavaCC allows multiple language definitions to exist in a single specification, switching grammars based on the detection of a special token in the input. This makes developing and maintaining each grammar difficult though, so instead we wrote a pre-parser to extract such embedded code into its own file. This allowed the language grammars to be written and maintained independently.

In cases where the embedded code is not clearly delineated in source code, it is often a string literal. We were not interested, at this stage in our work, in trying to analyze these strings to distinguish between plain text and embedded code, or to recognize that a nonsensical code fragment is a substring of a larger expression. To avoid this issue, the token definitions for the language were altered to split apart string literals based on non-word characters (e.g. whitespace, punctuation) and insert the resulting words back into the token stream. The tokens were tagged to indicate that they originated from a string literal. This gave us the flexibility to add productions to the island grammars to deal with the embedded code alongside productions for the enclosing language. The existing water productions (see Section 3.5) in the grammar otherwise ignored the new tokens.

**Example island grammar definition**

In Section 4.2.1, we presented a method declaration example from a Java class in JPetStore. The key syntax for a method declaration is the name of the method declaration, immediately
followed by an opening parenthesis. Within the parentheses can be a comma-separated list of parameters and their types. Finally, a set of braces come after the parentheses, that delineate the statements comprising the body of the method. Figure 4.2 shows an island grammar production we wrote to match this. \texttt{Reference} is a special production meant to capture identifiers and their types, and can match either one identifier token (in the case of the method name), or two such tokens (in the case of the method parameters). \texttt{AmbiguousReferences} is another production meant to handle the contents of the method declaration’s statement block.

The grammar production shown in Figure 4.2 however contains more detail than we just identified as being minimally characteristic of a method declaration. In practice we also need to manage visibility and behaviour modifiers that can prefix the method declaration, the return type, and any exception information provided. These are not important for estimating dependencies on method declarations, but must be dealt with since they are part of the Java syntax. We wrote a water production called \texttt{JunkUntil(\ldots)}, that accepts a “stop-token” as an argument and then discards tokens in the input stream until it reaches the “stop-token”. This allows us to elide unwanted syntax from Java, without having to write detailed grammatical productions.

4.3 Experimental Evaluation

We are interested in: (1) understanding how incremental improvements to the island grammar improves its effectiveness in detecting dependencies, and (2) observing how the effort nec-
necessary increases as we attempt to improve dependency detection. To empirically determine the cost-to-accuracy relationship between island grammars, we conducted a case study that applied separate configurations of Luther to an industrial software system. Each configuration of Luther supported a different level of semantic richness in its island grammars, and we compared the dependencies detected by each configuration against the actual dependencies manually determined to be in the system.³

OpenBravo⁴ (an open-source, web-based, enterprise management system) was chosen for our study. It comprises more than 3,000 files written in Java, HTML, XML, and SQL, totalling over 300,000 lines of code.

4.3.1 Methodology

Our approach for determining the cost-to-accuracy relationship between island grammar richness and dependency analysis involved configuring Luther for four different island grammars: lexical, keyword, nesting, and references. The characteristics of these grammars are discussed further in Section 4.3.2. Each Luther configuration was supplied a set of source code from the OpenBravo system, and the dependencies detected by Luther were compared against a set of known dependencies manually identified by the authors. The precision (fraction of results that are true positives) and recall (fraction of actual results that are returned) of each island grammar configuration was calculated by comparing its success or failure to match the actual results. The size of each grammar in lines of code (LOC) was used to approximate the cost of developing each grammar.

Actual Dependencies

To qualify the accuracy of dependency detection for each Luther configuration, we needed to know what dependencies existed in the source code. A base set of the dependencies for

---

³Additional details for Sections 4.3.1 and 4.3.2 are in Section C.3
⁴Version r2.20, Revision 438.
the source code was determined using a combination of automated and manual approaches. Because our process for determining true dependencies was extremely labour intensive, we restricted the case study to two files in the OpenBravo system: the VerticalMenu class and the Menu_data.xsql file in the org.openbravo.erpCommon.utility package. Menu_data.xsql is an XML tree containing SQL queries with mapping information tying each query to a method declaration, similar to the Account.xml example presented in Figure 2.3. The VerticalMenu class uses the a Java object representing Menu_data.xsql to generate a webpage when executed. Together, the two files comprise 429 non-commented LOC.

We first used Eclipse’s Java Development Tools and the JDOM\(^5\) XML parser to extract an initial set of tokens and their attributes from the Java, XML, and SQL source code. This initial set comprised 2009 identifiers (not including reserved words). The set was then refined by manually investigating the context of each token to exclude spurious tokens, add missing tokens, and provide additional semantic information in some cases to differentiate between tokens. Our investigation pared down this set to 867 semantically different tokens. This was the bulk of the work required.

To identify polylingual dependencies, we looked for instances where lexically similar tokens appeared in two or more languages. This required some modification of our extracted token data as SQL is case insensitive, and we frequently encountered cases where the same table or column name used different capitalization schemes. We also noticed that a token in Java would sometimes have a similar counterpart in SQL, but with additional underscores inserted into the identifier.

We examined the context of each of these tokens to determine if there was enough semantic information present to indicate a dependency existed between them. As an example, we present two code excerpts from the VerticalMenu.java and Menu_data.xsql files in the OpenBravo system. In Figure 4.3, the selectIdentificacion method is semantically de-

\(^{5}\)http://www.jdom.org, as of 2008/08/05
dependent on the value of the name attribute in the SqlMethod element in Figure 4.4. However, strCliente in Figure 4.3 is not semantically dependent on the value of the name attribute in the Parameter element in Figure 4.4. The tokens are lexically the same, and their naming does suggest an intent by the developer to imply that they are conceptually the same. But when we examined other SqlMethod XML elements in the same file, we noticed that the name attribute of their Parameter child element would often have a value completely different than the variable name supplied in the matching method invocation in the Java code. So we ruled this out as a valid semantic dependency.

In all, we identified 105 polylingual dependencies just within the two files. Our determination of the real dependencies between two files in the OpenBravo system took over a week.

MenuData[] data =
   MenuData.selectIdentificacion(this, strCliente);

Figure 4.3: Excerpt from VerticalMenu.java.

<SqlMethod name="selectIdentificacion"
   type="preparedStatement"
   return="multiple">
   <SqlMethodComment></SqlMethodComment>
   <Sql>
   SELECT ...
   </Sql>
   <Parameter name="strCliente"/>
</SqlMethod>

Figure 4.4: Excerpt from Menu_data.xsql.

4.3.2 Island Grammar Sets

Each configuration of Luther utilized three island grammars written for Java, XML, and SQL. Each grammar specification has three sections: (1) token definition, (2) production defini-
tion, and (3) support framework. Token definitions are supplied by the developer as regular-expression patterns, and the developer can specify if certain symbols should be ignored. Productions are written by the developer to describe syntax in the language he wishes identify. Each production consists of a pattern of tokens and other production patterns. The support framework provides common functions and helper methods for each grammar, such as the interface that serves as the entry point into the parser, and methods to store information in the symbol table.

**Commonalities across grammars**

We chose a pragmatic approach to writing each grammar that involved defining the simplest grammar specification first, and then incrementally improving the island grammar richness to achieve each successive version. Luther provides a template for JavaCC parser specifications that defines basic commonalities across grammars (e.g., C-style token definitions, helper methods for storing tokens in Luther’s symbol table) which can be tailored as needed. The template also specifies a grammatical production called `JunkUntil(...)` that takes a “stop-token” as a parameter, and consumes token input until this token is reached. `JunkUntil` is useful in many situations for eliding unwanted language syntax.

**Incremental grammar development**

The *lexical* grammar is the starting point for each language, and consists of only one production that accepts all “identifier” tokens.

The *keyword* grammar expands the token definition section with the reserved words defined for each language. We were able to largely cut-and-paste the reserved word definitions provided by the JavaCC grammars for Java 1.5 and PL/SQL. Aside from meta-data, XML does not define reserved words and as such the XML keyword grammar was identical to its lexical grammar.

The *nesting* grammar builds on the keyword grammar by recognizing tokens in each lan-
language that reflect a nesting structure. However, SQL was a special case. Normally in an SQL query, parentheses are used to represent parameter lists (as in `INSERT` queries) or nested queries where the query result acts as an input into its parent query. For static analysis, neither of these points are relevant: intermediate query results cannot be calculated until run-time and, in the case of `INSERT`, recognizing the parentheses does not aid in deciphering the intent of the statement. The SQL nesting grammar remained unchanged from the keyword grammar.

The *references* grammar builds on the nesting grammar by attempting to recognize the context of a token as indicating it is a reference to or a declaration of a symbol. Some specifics of how we handled this are described in Section 4.2.3. Each of the three languages required a difference approach here.

XML trees do not implicitly contain the concept of a declaration of a symbol; the existence of an XML element with a specific name constitutes an implicit declaration of the symbol. Provided that the namespace does not change, all occurrences of the same element name may be treated as a reference to a common identifier. In the files we analyzed, namespaces were not in use.

In SQL, provided all queries are executed within the confines of a single database, table and column names are treated as global references in queries and their declaration is implicit to the actual existing database. SQL does allow table and column identifiers to be renamed with aliases within a single query, so productions were added to understand the semantics of these special cases.

In Java, a declaration depends heavily on context. Because the main entry point into each Java source code file is a type declaration—typically a class—a top down approach to detecting declarations and references is taken. The initial production tries to match the `class` declaration, and passes the body of the class to another production which tries to distinguish between type and method declarations within the statement block. Up until this point, writing the island grammar productions is fairly straightforward since a limited number of major
Table 4.1: Island Grammar Detection of Polylingual Dependencies.

<table>
<thead>
<tr>
<th>Island grammar</th>
<th>Total matches</th>
<th>True ‘+’s</th>
<th>False ‘+’s</th>
<th>False ‘−’s</th>
<th>Size (LOC)</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
<th>Tradeoff ratio (% / kLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexical</td>
<td>2180</td>
<td>101</td>
<td>2079</td>
<td>4</td>
<td>630</td>
<td>4.6</td>
<td>96.2</td>
<td>7.4</td>
</tr>
<tr>
<td>keyword</td>
<td>1705</td>
<td>101</td>
<td>1604</td>
<td>4</td>
<td>861</td>
<td>5.9</td>
<td>96.2</td>
<td>6.9</td>
</tr>
<tr>
<td>nesting</td>
<td>1709</td>
<td>105</td>
<td>1604</td>
<td>0</td>
<td>872</td>
<td>6.1</td>
<td>100.0</td>
<td>7.0</td>
</tr>
<tr>
<td>references</td>
<td>37</td>
<td>37</td>
<td>0</td>
<td>68</td>
<td>1028</td>
<td>100.0</td>
<td>36.6</td>
<td>94.7</td>
</tr>
<tr>
<td>idealized</td>
<td>105</td>
<td>105</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>100.0</td>
<td>100.0</td>
<td>–</td>
</tr>
</tbody>
</table>

syntax productions are possible. By contrast, identifying major syntax within the body of a method declaration’s statement block is considerably more difficult. Writing productions to ignore unnecessary syntax in this case proved to be exceptionally difficult due to the variety of token combinations that can appear, and because the JavaCC’s top-down approach to grammar specifications required that we anticipate these combinations in our productions.

To overcome this obstacle, we essentially wanted to change the parser to use “bottom-up” recognition just within the bodies of method declarations. The idea would be to have the parser recognize tokens, check a limited number of patterns to see if it matched a known production and if not move on. To simulate this, we wrote a production called **AmbiguousReferences**, as seen in Figure 4.2. Like **JunkUntil**, it accepts a “stop-token” as a parameter and behaves as a bottom-up parser until that “stop-token” is reached. For Luther, we implemented **AmbiguousReferences** by hand in Java and integrated the code into the JavaCC template we used for our grammar specifications. We feel this is a specific technical limitation of the tools chosen for our work, and alternative remedies are discussed in Section 4.5.1.

4.4 Results

Table 4.1 presents the results for detecting polylingual dependencies in the source code. Each dependency detected is crossing a language boundary (e.g., from Java to XML). “Total matches” indicates the total number of dependencies identified by each grammar. The “true positives”
column shows the number of dependencies correctly identified by the grammar, while the “false positives” column records the non-existent dependencies reported. The “false negatives” for each grammar indicate how many real dependencies failed to be detected. Size reports the sum total length of all three island grammars written at that level. Precision and recall percentages were calculated based on the preceding data. Finally, the tradeoff ratio presents the cost-to-accuracy ratio as the percentage of precision achieved per kLOC written in the island grammar specification.

Generally, we see a gradual improvement in precision as grammar richness increases. The presence of four false negatives in the lexical and keyword grammar’s results is due to a peculiarity of the XSQL syntax used in OpenBravo which can be detected in the nesting grammar. Were it not for that, the nesting grammar would have made no improvement in precision over the keyword grammar. It may be the nesting grammar is an example of a island grammar improvement that provides little benefit.

The references grammar was able to achieve significant precision relative to the other grammars. The grammar has sufficient semantic understanding of the source code that it was possible to precisely identify certain specific types of dependencies. All 37 of its detected dependencies were found to be correct. However, this came at the cost of recall as almost two-thirds of all dependencies we identified failed to be detected. This was in part because of the information the references grammar discards, and also because the analysis heavily discriminated against potential matches which did not satisfy all of the dependency rules. By comparison, the lexical and keyword grammars have very little attribute information for each token and so are unable to discriminate their results further. The precision to cost ratio is quite high in this case, but hides the grammar’s failure to recognize most known dependencies in the system.
4.5 Discussion

The results of our case study suggest that island grammar approaches can offer some improvement over purely lexical approaches, but require a greater effort investment on the part of the developer tasked with writing island grammars. Further, as these island grammars increase in richness, developers may find that the dependencies they are seeking to detect are ignored by some of the island grammar productions they write. Developers may also decide to take a very restrictive approach to identifying dependencies between tokens to ensure high precision rates yet, find they have to reject more dependencies than would make such a tool useful. It may be that lower recall rates are acceptable if the ignored dependencies are not significant to the developer; a tool designer needs to carefully consider the tasks he wishes to support.

4.5.1 Limitations of the Study

Our approach and testbed have several limitations which affects the quality of our results.

Measurement

We acknowledge that using LOC as a measure of the cost to develop an island grammar specification is not a good indicator of effort. Certainly our own experience indicates that a grammar that is only slightly larger than another may still be significantly more complex and take longer to develop. Our choice of LOC as a cost measurement was due to the lack of a suitable alternative, and its ease of measurement. Our own development of the island grammars for Luther could not be used as an effort estimate, due to significant learning effects.

Study size

Our study’s restriction that we manually verify and refine a base-set of dependency information to compare to the testbed results forced us to drastically limit the size of our study. Our precision and recall values for each grammar would likely change if applied to other files or
the entire system (assuming we still knew the actual dependencies). However, the relative performance difference between grammars will likely be similar in magnitude in these cases.

**Token transformations**

The Luther testbed and our island grammar design relies heavily on lexical exactness of two tokens as a first step in inferring a dependency between them. Some polylingual frameworks (like JNI) will apply prefixes or other changes to tokens in each language rendering them lexically similar, but not identical, and thus unable to be detected by Luther. This is a current technical limitation of our approach, and could be solved by supporting the definition of token transformation rules so that dissimilar tokens could still be matched as being dependent on each other.

**Parser technology**

Some problems we encountered in developing island grammars may be specific to JavaCC or LL(k) parsers. Previous work used a Generalized LR parser [Moo01] for writing island grammars; it is unclear whether this would display a marked improvement over our results. We chose JavaCC for use in Luther because it is a viable, industrial-strength approach with which we were familiar.

### 4.5.2 Future Work

Luther is currently a testbed meant for empirically studying the accuracy of island grammars. In future studies, we would like to address some of the limitations described in Section 4.5.1. A large-scale study would provide more insight into the cost-to-accuracy relationship, but would need to compare the testbed’s results against the best available tools as manual identification would be impractical. We would also like to see if a change in parser technology can provide a significant effort savings when creating island grammars.

We also want to evolve Luther into a tool to support developers writing dependency de-
tection tools using island grammars. The key obstacles to overcome in the current version of Luther are those impacting the ease of use by software developers. For example, the existing mechanisms for defining token attributes and dependency relationships require programmatic specification, requiring significant effort to reuse. A better approach would be to allow developers to write simple specifications to describe token attributes and dependency rules; using these specifications, Luther could then generate code to implement the specifications within its framework.

4.6 Summary

Our approach applies island grammars for detecting dependencies in polylingual software systems. We created a testbed called Luther to analyze the cost-to-accuracy relationship between a series of island grammars. Our experimental evaluation involved configuring Luther with four different island grammars, and using each configuration to analyze static dependencies in the OpenBravo system. The results for this study suggest that the effort-cost in writing more accurate island grammars rises faster than the resulting accuracy.
In this chapter I describe GrammarSketch, a lightweight approach to providing approximate
dependency analysis tool support for polylingual software systems. GrammarSketch is de-
digned to be configured by a single developer for the particular software system they work
on, but this configuration information is easily shared with other developers in their team or
organization. Once configured, the tool highlights points in the source code where a depen-
dency has been found, shows the developer what other portions of the system are dependent
on that point, and facilitates navigation between dependency locations so that a developer may
examine dependency locations easily.

The rest of this chapter is organized as follows: Section 5.1 summarizes the key findings
from the Luther testbed that I wanted to apply in future work on polylingual dependency analy-
sis. Section 5.2 describes the architecture and design of the GrammarSketch tool. Sections 5.3
through 5.5 cover the usage of the tool: Section 5.3 details how developers configure the tool by
writing simplified regular expression patterns, Section 5.4 explains how GrammarSketch ex-
tracts identifiers from the source code and infers the presence of dependencies, and Section 5.5
describes how the developer uses the configured tool in their work.

5.1 Introduction

It was my intention to expand the Luther testbed into a full fledged framework for generating
tool support for developers; however, while the tool I present in this chapter, GrammarSketch,
is the “spiritual” descendant of my previous research and study described in Chapter 4, it is not
based on island grammars, but on the regular expression techniques described in Section 3.3.
In this section, I outline how my work with Luther lead to the creation of the Grammar-Sketch tool, and what difficulties along the way caused me to change the technique I chose to base my tool support on. In Section 5.1.1, I describe two key results that I took away from the Luther case study that I felt were important for developing future tool support, and in Section 5.1.2 I explain the issues and difficulties I had with the island grammars technique that caused me to later base GrammarSketch on regular expressions.

5.1.1 Lessons Learned From Luther

The Luther testbed case study on the OpenBravo ERP software system discussed in Section 4.3 yielded important feedback on configuring partially-semantically-aware tool support; at the time of publication of that research, I was exploring ways to expand Luther into a framework for generating polylingual dependency analysis tool support. The key issues I wanted to address were:

1. The cost of each improvement to the semantic awareness of the Luther testbed, relative to its accuracy in detecting polylingual dependencies, seemed to diminish at each stage. While I reported the relationship of accuracy to effort in terms of lines-of-code (LOC) in Section 4.4, the difficulty of writing the more sophisticated island grammars is not adequately expressed in that measurement; the complexity of developing semantically rich island grammars is daunting, even for a specialist. Some of the difficulty here may rest with the decision to use JavaCC; other researchers have also expressed frustration with its notation scheme [Moo02].

2. It was difficult to balance precision and recall in dependency detection. As Luther transitioned from a lexically-aware to semantically-aware understanding of the code, there was an abrupt shift: from high recall and low precision in the lexical spectrum, to high-precision, and low recall in the semantic.
5.1.2 Shifting From Island Grammars to Regular Expressions

The name I coined for this new research prototype tool I was developing was *GrammarSketch*. I employ the metaphor of a “sketch” in describing how this tool is used for polylingual dependency analysis: a sketch is a drawing that bears a crude resemblance to what it is attempting to portray, but is easier to draw than a detailed portrait, while remaining suggestive enough to convey the intended image to its viewer. With GrammarSketch, I aimed to create a tool that a developer could use to “sketch” what a particular programming language looked like, by filling in only those key semantics essential for dependency analysis. The resulting “grammar sketch” would be missing numerous details (which I could compensate for using island grammars), but should bare enough of a resemblance to the language it attempts to describe to actually be useful for dependency analysis.

After investing significant effort in researching how to adapt the Luther testbed to address the points highlighted in Section 5.1.1, I decided to abandon the testbed platform I had created. In the end, the JavaCC parser generator on which Luther was built, and for which I previously had difficulty applying to island grammars (see Section 4.5.1), proved to be too rigid to adapt for my purposes. Rather than change to another parser technology platform and continue using island grammars, I decided to instead base my tool support on lexical analysis techniques, while still incorporating some of the concepts used by island grammars to simplify the notation.

When kept simple, regular expressions are very easy to use. But, as we try to increase a pattern’s richness to improve its accuracy, the resulting expression (as shown in Section 2.3) becomes unreadable, unmaintainable, and error prone. However, re-developing the GrammarSketch prototype to be based on regular expressions looked to be more feasible than the prospect of researching how to incorporate a new, unfamiliar parser technology platform into GrammarSketch to continue using island grammars. I judged that the problems I previously described with applying regular expressions to polylingual dependency analysis could be overcome by incorporating some ideas gleaned from my work with Luther and island grammars.
I adopted an approach similar to the LSME tool (presented in Section 3.3), where the specifications necessary for configuring analysis tool support were split between defining lexical patterns to match in the source code, and an “action language” to defined how pattern matches were to be combined to generate a source code model. Based on the results of the Luther testbed, I chose to focus GrammarSketch to specifically recognize the same semantics that comprise Luther’s *references* grammar. In doing so, I was able to devise mechanisms to keep the regular expressions written by developers for the tool simple: I tailored the syntax used to configure GrammarSketch to recognize just these specific semantic patterns, and provided constructs gleaned from island grammars (specifically, the use of “water” patterns) that could later be used to transform a pattern in this simplified notation, into a full-fledged regular expression with enough sophistication to accurately recognize the intended syntax. Further, by focusing GrammarSketch on recognizing only a predetermined set of semantics, I also was able to obviate the need for an action language; instead, I predefined how these semantics could be combined to infer the presence of polylingual dependencies.

### 5.2 The GrammarSketch Tool

Figure 5.1 is a screenshot of the GrammarSketch dependency analysis tool, installed as a plugin to the Eclipse Integrated Development Environment (IDE). The GrammarSketch tool is intended for use by developers looking to investigate dependency relationships in a polylingual system, where a reasonable approximation of the dependencies is sufficient for their needs. The tool is configured with a set of simple patterns to recognize the syntax of key semantics in each programming language in the software; from this configuration information, GrammarSketch analyzes the source code of the system and predicts where dependencies exist.
5.2.1 The Architecture of GrammarSketch

Figure 5.2 is an overview of the GrammarSketch tool. GrammarSketch is a plugin to the Eclipse IDE\(^1\). A developer using the plugin will have a project in Eclipse with the complete set of source code for the software they intend to analyze. The developer writes a set of patterns using the GrammarSketch notation (see Section 5.3) to describe key semantics for each language in their software system, and stores the patterns in a special file inside their software project (the file is currently stored at `gsk/grammarsketch.gsk`). GrammarSketch compiles these patterns into an equivalent series of regular expressions, and associates each pattern with the

\(^1\)http://www.eclipse.org/, version 3.3.2 is currently supported
Figure 5.2: Architecture of the GrammarSketch plugin.

GrammarSketch then traverses the source code stored in the developer’s software system, and determines for each file which programming language(s) are in use based on the file’s extension (e.g., source code files ending with .java will consist of Java source code). For each source code file in the system that GrammarSketch recognizes, the tool applies the compiled patterns appropriate for the language encountered, and extracts the matches found into a symbol table that tracks where each match was found, the context where the match was made, and the pattern that detected the match.

Once GrammarSketch is finished traversing the source code, the symbol table is analyzed to determine where dependencies are present in the source code. At this point, GrammarSketch begins to interface with the source code editor windows in which Eclipse displays the developer’s source code, as seen in Figure 5.1, and highlights those identifiers in the source code where it has determined that a dependency is present.
5.2.2 Simplifying Configuration Through Imposition

The feedback of the various island grammars used to configure the Luther testbed suggested that the references grammar used in the testbed provided the most significant improvement over the more lexical grammar configurations used, but the effectiveness of the island grammar was offset by the difficulty of writing the appropriate grammar patterns. GrammarSketch incorporates this feedback into two core design decisions:

1. the tool presumes which semantics are important to recognize for dependency analysis, specifically declarations, types, and references, and is configured to find dependencies based on recognizing these semantics in source code; and,

2. developers need only to write patterns to match the syntax necessary for recognizing just the above semantics in each language under analysis.

By imposing on the developer which semantics should be recognized, GrammarSketch reduces the effort needed from the developer to configure the tool for each language in their software system: developers will only write those patterns necessary to recognize these semantics, and the notation language used to describe the relevant syntax can be tailored specifically to this task.

5.3 The GrammarSketch Notation

GrammarSketch relies on the developer to define the patterns needed to recognize relevant syntax in the source code. This allows the tool to be flexibly configured for the arbitrary combinations of languages that may exist in the developer’s software. However, the results of the case study in Section 4.5 emphasize that the effort required from the developer to configure a tool should be as low as possible: it seems reasonable to expect that a developer could explain
or identify the general structures of a language, but it would be difficult for them to describe it in a notation like that of JavaCC, or even as a regular expression (see Section 2.3).

This section outlines the GrammarSketch notation that developers use to specify the relevant syntax patterns in source code. I discuss the underlying principles of the notation in Section 5.3.1, Section 5.3.2 specifies the syntax of the four main productions in the notation, and I end in Section 5.3.3 by revisiting the regular expression example in Section 2.3, to show how much simpler the same pattern expressed in the GrammarSketch notation is.

5.3.1 Principles of the GrammarSketch Notation

GrammarSketch borrows several concepts and ideas from work on “fuzzy parsing” [Kop97], island grammars [Moo01, Moo02], and the results from my previous work with Luther (see Chapter 4), to inform the design of its notation.

The pattern notation should aim for human readability

Describing what a method declaration looks like in Java can be done in an ad-hoc fashion by any developer familiar with the language, and the explanation will likely be easily understood by another developer; describing that method declaration as a regular expression (as in Section 2.3), or using a formal grammar notation like the Backus-Naur Form is more difficult, and hard to communicate to another developer who does not share the necessary expertise with the notation. To address this, the GrammarSketch notation acts as a syntactic sugar for regular expressions: it abstracts several core patterns that are common to C-style programming languages and that are important for dependency analysis, and represents them using sensibly named identifiers that aim to self-document the concept they match.

Eliminate details that hinder the expression of intent

Many programming languages provide a number of conveniences in how code can be written that in turn make writing tool support for analyzing the code much more difficult. For example,
the flexibility that a developer has in how they use whitespace in writing their Java code makes it difficult to write complex regular expression patterns correctly [AG06]. The intent of the resulting pattern becomes obfuscated by the special cases that need to be addressed. GrammarSketch instead tries to anticipate these details, where possible, for the user. The goal is that the patterns written can clearly delineate what the developer intends to match, rather than be cluttered with non-trivial but irrelevant detail.

Allow implicit and explicit ignoring of code

The island grammars technique discussed in Section 3.5 is predicated partly on the assumption that the portions of source code we are interested in are a small fraction of the complete code base [Moo01]. Island grammars approach this problem by designating special grammatical productions called “water” to ignore sections of source code.\(^2\) The choice to base the GrammarSketch notation on regular expressions allows us to implicitly ignore source code not matched by any patterns that a developer described. However, GrammarSketch also provides an explicit means to ignore any code found between specific symbols in the code, so that developers can easily ignore irrelevant syntax.

Use “anchor points” to help deal with ambiguity

“Fuzzy parsing” describes a category of techniques that attempt to correctly parse source code in situations where there is ambiguity in the syntax [Kop97]: for example, the Java syntax `java.lang.String` could mean

- The fully qualified name of the `String` class,
- A field access (`String`) on an object stored in the `lang` field on a `java` object,
- Or several other potential variations on points one and two.

\(^2\)This may seem like an obvious step, as regular expression patterns by default ignore all text that they cannot match. But a grammar is expected to recognize all strings that occur in a language, and so must understand to some degree syntax that a developer considers to be irrelevant.
Language grammars resolve this issue by specifying all the possible combinations that may occur, and using “look-ahead” techniques to grab additional context surrounding the syntax to resolve the ambiguity. Fuzzy parsing techniques instead attempt to resolve such ambiguity with less information, by looking for specific symbols called “anchor points” in the syntax. The presence or absence of an anchor point dramatically reduces the possible meanings for ambiguous syntax.\(^3\)

Regular expression patterns implicitly allow the specification of anchor points; the GrammarSketch notation makes the specification of anchor points explicit, and marks such symbols as useful only for finding matches. The portions of the source code matched by an anchor point are ignored when identifiers are extracted to determine the presence of dependencies.

5.3.2 The GrammarSketch Notation

The GrammarSketch notation comprises four main components, that are combined by developers to describe the syntactic patterns in source code which describe those semantics relevant for GrammarSketch. These components are:

**GSK-Type and GSK-Reference**

GrammarSketch assumes that recognizing *declarations*, *types*, and *references* in each programming language under analysis is a sufficient basis to approximate the real polylingual dependencies present. The focus of the GrammarSketch notation is primarily on recognizing those identifiers found in pattern matches in the source code, that represent these semantics.

GrammarSketch conflates the semantics of *declarations* and *types* into what I will refer to as a *GSK-Type*, denoted by the keyword `type`. A GSK-Type refers to any identifier in the code whose meaning has a global significance to the system; this approximation of the semantics

\(^3\)In the last paragraph of Section 4.3.2, I describe how Luther used a special production called Ambiguous-References to match patterns in the bodies of source code. The technique described in the paragraph is in fact fuzzy parsing; at the time of publication of that research, I was not aware that other researchers had previously investigated this issue and had also devised similar solutions.
of a type and a declaration is useful for languages like XML and SQL where the semantics of
declarations may be implicit, and also provides a crude approximation of scope resolution.

Reference semantics are approximated by what I will refer to as GSK-Reference, denoted by
the keyword reference. A GSK-Reference refers to any identifier in the code whose meaning
has a local significance to the system; GrammarSketch interprets this as having no significance
outside the file in which the match is discovered. Again, this provides a crude approximation
of scope resolution by allowing patterns to restrict in what ways GrammarSketch is allowed to
resolve dependency matches on that identifier.

Anchors
GrammarSketch allows the specification of anchor-points (see Section 5.3.1), referred to as
anchors, in patterns to improve discrimination. The syntax of an anchor involves wrapping
any character or string that constitutes the anchor-point, with single quotes (e.g, ’public’,
or ’{‘). Anchors allow the specification of syntax that the pattern must match to be correct,
but which GrammarSketch will subsequently ignore when determining which portions of the
pattern are relevant for dependency analysis.

String literals
It is quite common for code, or important identifiers, from a programming language to be
embedded in another language, typically as a string literal. It is sometimes necessary to attempt
to match patterns inside of such string literals, or to capture the entire contents of the string.
GrammarSketch provides two mechanisms for dealing with this:

• Since most string literals are delineated with double quotes, the developer can specify
each double quote as an anchor, and then write how patterns inside the quotes should be
matched.\(^4\)

\(^4\)During the case study, a bug was found in GrammarSketch where double quotes would not always correctly
work in indicating the presence of a string literal. The second mechanism, the STRING_prefix, works correctly
and is used instead at the moment. The bug is being addressed.
• The developer can also add a unique prefix, STRING_, to the type and reference syntax described earlier. This instructs GrammarSketch to grab the entire contents of the string, and treat it as either a GSK-Type or a GSK-Reference, respectively.

The Junk production

Some patterns can be difficult to specify in detail, because of the various syntax combinations that may exist in the source code. Rather than anticipate all such combinations, if the developer deems that the information in these syntax variations is less relevant, they can use the Junk production ignore everything GrammarSketch finds at that point in the pattern.

The syntax of the Junk production is [junk], followed by an anchor that represents the terminal for that production. The Junk production accepts every character it encounters until it reaches the terminal specified by the anchor.5

5.3.3 Example: Using GrammarSketch to Match Method Declarations in Java

In Section 2.3 I attempted to show how difficult it was to write a regular expression pattern to match a method declaration in Java. To show how the GrammarSketch notation is used, and improves on the equivalent regular expression pattern, I will revisit that example.

The original goal was to match a method declaration in Java, such as in Figure 5.3. Originally, I decided that I wanted to capture the method’s name, and the return type. That meant having to write:

\(([^a-zA-Z_]*)\s+([^a-zA-Z_]*))\)

5This can lead to some problems: GrammarSketch is greedy in its pattern matching. If a pattern using a [junk] production finds an initial match at the start of the file, but the first anchor it matches is near the end of the file, GrammarSketch will assume the pattern matches the entire file, even though this is likely not what the developer intends. In practice, this does not prove to be a significant problem; while the entire match may be extracted, only the portion of the match designated by the type or reference syntax are in fact extracted for dependency analysis. This does however contribute to a loss of precision in the tool, and could be improved by the application of size heuristics to reject overly large matches (Murphy uses this approach to solve a similar problem with the LSME tool [MN96]).
public void doPost (HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
    VariablesSecureApp vars = new VariablesSecureApp(request);

    if (vars.commandIn("DEFAULT")) {
        printPageDataSheet(response, vars, "0", false);
    } else if (vars.commandIn("ALL")) {
        printPageDataSheet(response, vars, "0", true);
    } else
        throw new ServletException();
}

Figure 5.3: Example code snippet.

Now, I can write:

type type

Using \texttt{type} here tells GrammarSketch I am looking for two consecutive C-style identifiers, and these identifiers should be treated as GSK-Types. I have not explicitly stated anything about the whitespace that may exist between these two identifiers, but this is not an issue as GrammarSketch understands that there could be any amount of whitespace between the two identifiers and deals with it appropriately.

The next step I took was to try and see if there was a parameter list after the method declaration. If the parameter list was not important for my analysis, I could simply have the pattern try to detect two consecutive identifiers in front of a parenthesis:

type type '()'
Or I could have the pattern recognize the set of parentheses while ignoring the parameter list inside:

\[
\text{type type } '(\ [\text{junk}]\ )')
\]

In the original example, I decided that I wanted to capture the parameter list. This is how it looked like as a regular expression:\(^6\)

\[
([a-zA-Z_\w]*)\ \s+\ ([a-zA-Z_\w]*)\ \s+\ "\ \s+\ (([a-zA-Z_\w]*)
\s+\ ([a-zA-Z_\w]*)\)?\ \s+\ "\ \s+\ ([a-zA-Z_\w]*)\ \s+
([a-zA-Z_\w]*)*\ \s+\ "\)
\]

Here is the GrammarSketch equivalent:

\[
\text{type type } '(\ (\text{type reference })?\ (',\text{type reference })*')
\]

Here, the \textit{reference} syntax denotes that the parameter names in the method declarations should be treated as GSK-References, or having a meaning significant only in the file the pattern is matched. This is a larger scope than these identifiers should have (their scope should only apply to the body of the method declaration they are associated with), but it is still a useful approximation to make: the restriction prevents GrammarSketch from wrongly matching the identifier to other, lexically similar but semantically different identifiers in other files, without the need for requiring the developer to specify additional syntax to define the scope of the reference.

\(^6\)During my defense, one of my examiners pointed out that this regular expression actually does not match the code as presented - the \texttt{\s+} after the the " should instead be a \texttt{\s*} so that it does not require a space to be present after the opening parenthesis in a method declaration. In fact, to correctly match the Java method declaration syntax, the \texttt{\s+ before and after} the " should be \texttt{\s*}. Rather than fix this occurrence, I chose to leave it in - partly in homage to Jörg Denzinger who caught this, but also to highlight how problematic it is to write these kinds of regular expressions, and how easy it is to make a small but crucial mistake in the syntax.
Finally, in the original example I recognized the opening brace of the method declaration, while also a potential list of exceptions associated with the method declaration. The final regular expression pattern looked like this:

```
([a-zA-Z_\w]*) \s+ ([a-zA-Z_\w]*) \s* "\" \s* (([a-zA-Z_\w]*) \s+ ([a-zA-Z_\w*])? (\s* \"\", \" \s+ ([a-zA-Z_\w*]) \s+ ([a-zA-Z_\w*])\)* \s* \"\")\" \s* \"\{\"
```

The GrammarSketch equivalent:

```
type type '('(' (type reference)? (',', type reference)* ')')'
[junk] '{'
```

While the pattern is complete, GrammarSketch also asks the developer to supply a few pieces of metadata about the pattern:

- The tool assumes that every pattern written has a specific language in mind, and asks that the developer associate the pattern with a specific programming language. The current configuration of the tool supports Java, XML, and SQL files, but this is easily modified at will. GrammarSketch maintains a list of known file extensions that are mapped to programming languages (e.g., files ending in .java are associated with the Java programming language) so that the correct patterns are applied when analyzing source code by checking the extension of each source code file.

- The developer is also asked to provide a descriptive name for each pattern. The name is used for two purposes:

  1. It provides some contextual information to remind the developer of what the pattern was intending to match.

[7] The other case, where I tried to capture the types of exceptions thrown, is left as an exercise to the reader. Hah! Always wanted to say that.
2. When GrammarSketch applies the pattern, it associates the pattern’s name with the identifiers retrieved from the source code. In Section 5.5, I will show how GrammarSketch lets the developer know which patterns were responsible for detecting a particular match as a means of providing feedback as to what is working.

The pattern was written for the Java programming language, and I will call it “SimpleMethodDeclaration”. The final pattern looks as follows:

```
Java:SimpleMethodDeclaration:->
type type ') ( type reference )? ( ',' type reference )* ')'
[junk] '{'
<-  
```

5.4 Extracting and Analyzing Pattern Matches for Dependencies

When a developer activates GrammarSketch to display the dependencies present in a source file, GrammarSketch compiles the patterns into regular expressions, and applies them to all the source code in a software system. The tool traverses the source code, determines which patterns need to be applied to each source file based on its extension, and extracts those portions of the code that match the patterns, into an XML file. Once this process is completed for the entire code base, the XML file storing the matches is analyzed to build a symbol table listing the unique identifiers found during the analysis, the relationships between the identifiers in the system, the contexts each identifier is found in, and what patterns discovered that identifier.

---

8The reader may wonder what the significance of the colons and the arrows are in the pattern. My original intention was provide a graphical interface for GrammarSketch through which the developer would write the pattern, and it would be stored in an intermediate format. I decided later to abandon this approach as the underlying technology I was using (the JavaCC parser generator) was proving to be wholly unsuitable for what I needed. I did reuse much of the original framework however, so the addition of the colons and arrows are actually artifacts left over from an earlier approach that was otherwise abandoned.
5.4.1 Compiling the GrammarSketch Patterns

All GrammarSketch patterns written by a developer are stored in the developer’s source code project, in a file called /gsk/grammarsketch.gsk. GrammarSketch applies a parser to this pattern file written using the JavaCC parser generator to transform the notation into a set of regular expressions. The transformation rules are as follows:

- Occurrences of `type` and `reference` are expanded into a regular expression pattern appropriate for matching C-style identifiers, then wrapped in parentheses. This allows back-referencing to be used to recover the identifiers matched in the pattern.

- Occurrences of `STRING_type` and `STRING_reference` are replaced with regular expression patterns that match the entire contents of a string literal.

- Anchors are embedded into the regular expression pattern; however, GrammarSketch must check to see if the anchor matches the syntax of a regular expression and, if so, properly escape the anchor.

- Some regular expression syntax in the GrammarSketch pattern is carried forward into the transformed pattern without alteration. Currently, only the symbols used for repetition (i.e., ?, +, *) are supported.\(^9\)

- The presence of whitespace is dynamically calculated based on the combination of productions in the GrammarSketch patterns, according to patterns appropriate for C-style syntax languages.\(^10\) While the rules are simple, in practice they prove to be difficult to

---

\(^9\)GrammarSketch currently has a limitation with the use of the regular expression operators +, *. The tool relies on back-referencing support in Java’s regular expression engine to capture the `type` and `reference` portions of a match in the code. In many cases, the developer will wish to apply the +, * operators to a GrammarSketch pattern to capture a repeated series of `type` and/or `reference` identifiers in a pattern. However, the Java regular expression engine does not currently support back-referencing in the manner I need: the Java engine will only capture the last set of `type` and/or `reference` identifiers in such a pattern. I am currently exploring means of addressing this limitation in the tool.

\(^10\)There are several programming languages, such as Python and Ruby, for which whitespace is part of the language syntax (e.g., in both languages, the indentation level of a specific line of text has semantic significance). GrammarSketch is not yet appropriate for those languages.
apply:

1. No spaces are added to the start or end of the pattern.

2. If two identifiers are consecutive (e.g., \texttt{type type}, \texttt{type reference} etc.), they must be separated by at least one space, possibly more.

3. If the \texttt{[junk]} production is used, the production by definition matches all characters up until the anchor terminal. As a result, there is no need to add any spaces between the \texttt{[junk]} production and the anchor.

4. Between any anchor and any other GrammarSketch syntax element, zero or more spaces may be present.\footnote{This currently causes a problem in some situations. The developer may specify a series of string literals as an anchor pattern e.g. \texttt{`public` `void`}. In the translation, GrammarSketch identifies this as two consecutive anchors, and indicates that zero or more spaces may exist between them. Unfortunately this would mean in this case that \texttt{publicvoid} would be matched, even though that is not the developer’s intent. Currently, GrammarSketch is not smart enough to analyze the anchor patterns and realize that because these are consecutive string literals, they really should be separated instead by at least one or more spaces.}

5. Between any two other GrammarSketch syntax elements, there may be zero or more spaces present.

5.4.2 Applying the Patterns Against the Source Code and Extracting Identifiers

GrammarSketch currently uses the regular expression engine provided with the Java 6.0 Standard Edition SDK.\footnote{http://java.sun.com/javase/reference/api.jsp, as of 2008/05/05} The collection of transformed patterns are passed to a “matcher” class that manages the application of Java’s regular expression engine to the source code. For any arbitrary source code file, the matcher checks which patterns are appropriate for the language associated with the file, and applies those patterns to the code. Each portion of the source code matched is extracted to a \textit{symbol table}: an XML file that keeps track of all matches found so far (see Figure 5.2).
For each match, the symbol table stores the match’s location, the name of the pattern that found the match, the context of the match (i.e., all text matched by the pattern including those portions not relevant for dependency analysis), and the identifiers in the match tagged as GSK-Types or GSK-References.

GrammarSketch also does some bookkeeping during this extraction process: one weakness of the Java regular expression engine is that it does not identify the line and column positions where matches start and end. This is problematic; without such information, a developer will not know where the match occurs, or be able to navigate to dependencies detected in the source code. GrammarSketch tracks line break characters as they occur in the input stream as each file is loaded, and uses this information to determines the positions of matches found.

**Dealing with Pattern Collisions**

It is possible a developer will write two or more patterns that overlap; this may be because the pattern is poorly defined, or it may reflect the difficulty of writing patterns for that programming language’s syntax. This “pattern collision” may cause the same identifier in the source code to be recognized by multiple patterns, and may also mark the identifier as both a GSK-Type, and GSK-Reference. GrammarSketch deals with such collisions two ways:

1. GrammarSketch adds all of the matched patterns to the meta-data associated with that identifier; when the developer investigates what pattern recovered that particular identifier, they will see a list of multiple patterns.

2. In cases where an identifier is matched as both a GSK-Type, and GSK-Reference, the identifier is “promoted” to be a GSK-Type.

5.4.3 Conducting the Dependency Analysis

GrammarSketch makes several assumptions about how polylingual dependencies can be recognized, and based on these assumptions creates a set of rules that govern whether a dependency
is expected to exist between any two points in the source code. All of these rules must be satisfied for GrammarSketch to predict that a dependency exists.

1. A dependency may only exist between two identifiers.

2. Two identifiers must be lexically identical, ignoring issues of case, for a dependency to exist between them.

3. If the identifier is a GSK-Type, it is dependent on all other occurrences of that identifier which are also marked as being a GSK-Type, regardless of where they occur.

4. If the identifier is a GSK-Reference:
   
   • it is dependent on all other occurrences of that identifier which are also marked as being a GSK-Reference, provided those identifiers are in the same source code file; and,
   
   • if the pattern that is responsible for recognizing the GSK-Reference identifier also recognized a GSK-Type, and that GSK-Type preceded the GSK-Reference, the GSK-Reference is dependent on that GSK-Type.\(^{13}\)

5.5 Using GrammarSketch to Identify and Trace Relationships in Source Code

To show how GrammarSketch is used, and assists developers in investigating dependency relationships in source code, I present a walkthrough of how a developer would configure GrammarSketch for their software system, and then use the tool to assist their own investigation. The hypothetical developer in this walkthrough is using the Eclipse IDE, and has already installed the GrammarSketch plugin.\(^{14}\)

\(^{13}\)In cases where multiple GSK-Types are detected, the GSK-Reference is deemed to be dependent on that GSK-Type that most closely precedes it in the source code.

\(^{14}\)GrammarSketch is hosted on the web at http://pages.cpsc.ucalgary.ca/ cossette/GSK2008/update-site. To install the plugin, a developer would supply Eclipse’s “Software Update” manager with the previous web-address; Eclipse will then automatically download and install the plugin for the developer.
The developer’s first step will be to provide some configuration information for GrammarSketch, by defining the patterns GrammarSketch needs to recognize. For this example, the developer is working with a portion of code from the OpenBravo ERP system, which is discussed further in Section 4.3. The portion of the system they are interested in uses Java, SQL, and XML. If another developer has already configured GrammarSketch for the same software, the developer can simply copy the configured `/gsk/grammarsketch.gsk` file from the other developer into their own workspace, and then modify the patterns further if they wish. Otherwise, the developer creates their own `/gsk/grammarsketch.gsk` file and proceeds to define the patterns they are interested in, as shown in Figure 5.4. Details as to how these patterns are created can be found in Section 5.3.2.

![Figure 5.4: Writing GrammarSketch Patterns in the /gsk/grammarsketch.gsk file.](image)

At some point, whether to see how well their current pattern set works, or because the developer feels they have written a reasonably complete set of patterns, the developer will want GrammarSketch to show the dependencies it detects. The developer right-clicks on a source code file they are interested in, and selects the option “GrammarSketch: Show All Matches”, as shown in Figure 5.5. GrammarSketch will then open this source code file (if not already open), and highlight in the file every identifier that it believes is either a type or a reference,
based on the patterns written by the developer, as shown in Figure 5.6. Further detail as to how GrammarSketch applies the patterns written by the developer to the source code, and detects which identifiers form the basis of a dependency are discussed in Section 5.4.

Figure 5.5: Activating GrammarSketch to show matches for the patterns written.

GrammarSketch uses two mechanisms for highlighting: if an identifier has dependencies only within the same programming language (e.g., the identifier is a Java type, and all dependencies on this type are only found in other Java files) the identifier is highlighted in yellow, as seen in Figure 5.6. If the identifier has been matched to at least one dependency involving an identifier in a different programming language, the identifier is highlighted in purple to indicate that it is a polylingual dependency. While the current screenshots do not show it, if the developer chooses to hover their mouse cursor over a highlighted identifier a tooltip will pop up that informs the developer which patterns in their /gsk/grammarsketch.gsk file were

15Tooltips in Eclipse disappear as soon as any key is pressed, and to take a screen capture I need to use a combination of keystrokes.
Figure 5.6: GrammarSketch highlighting pattern matches.

responsible for finding this match. Based on the highlighting in the file, the developer can get a sense as to the accuracy of the GrammarSketch patterns they are using, but may also jump straight to those dependencies that are polylingual.

When the developer sees a potential dependency that interests them, they may choose to “drill-down” or see what other parts of the source code may be dependent on this identifier. To do this, the developer right-clicks on the highlighted identifier, and selects “GrammarSketch: Find Polylingual Dependencies” as shown in Figure 5.7. GrammarSketch then brings up a window at the bottom of the developer’s screen, and populates a table with all of the dependencies connected to the selected identifier, as in Figure 5.8. Each entry shows the identifier matched, its location, and the context surrounding the match. They can leverage this information to initially discriminate between the matches, and find those dependencies more relevant for their investigation task. An explanation as to how GrammarSketch determines the existence of dependencies between identifiers in the source code can be found in Section 5.4.3.
If the developer sees a dependency in the table that they wish to explore, they can double-click on it. In doing so, GrammarSketch will locate and open the appropriate source code file in a new editor window, and then jump to that dependency location (as seen in Figure 5.9). At this point, the developer may continue to explore other dependencies in the list, they may return to the original file they were investigating and inspect another identifier, they may examine what other dependencies may exist in this new file they opened, or they may open another source code file in the system to see what identifiers were found by GrammarSketch there.

5.6 Summary

GrammarSketch is a lightweight approach to providing approximate dependency analysis tool support for polylingual software systems. The GrammarSketch tool is designed around a set
Figure 5.8: List of identifiers dependent on the selected identifier.

of assumptions as to which semantics in general programming language are important to recognize for approximate polylingual dependency analysis support. GrammarSketch provides a simplified, regular expression-based notation, tailored to the dependency semantics the tool recognizes, and uses concepts from island grammars so that developers can “sketch” these semantics in a simple, human readable syntax; GrammarSketch can then expand those developer supplied patterns into complex and detailed regular expressions, suitable for recognizing dependency semantics in the languages in use in their software system. Once configured, the tool assists developers in conducting dependency analysis by identifying dependencies, and supporting the developer’s navigation between dependencies as they investigate a software modification task.
Figure 5.9: “Drilling-Down” a dependency.
Chapter 6

Evaluation and Results

In this Chapter, I present a case study, conducted with four participants, in which they configured the GrammarSketch tool for dependency analysis on the open source software system OpenBravo ERP. The case study was intended to evaluate the claims of the thesis in Section 1.4.1 The chapter is organized as follows: Section 6.1 outlines what the evaluation of my work aims to address, and poses two research questions. Section 6.2 explains the methodology I used to answer those questions, and Section 6.3 contains the results of each participant’s work in the study. In Section 6.4, I summarize the key findings from the case study, and use them to answer the research questions posed in Section 6.1.

6.1 Goals of the Evaluation

The two principle claims of my thesis statement (see Section 1.4) can be summarized as:

1. A single developer can easily configure the GrammarSketch tool for dependency analysis in a polylingual system.

2. The effectiveness of GrammarSketch would be at least comparable to alternative lexical approaches.

There is a tremendous amount of diversity in the problem space that the GrammarSketch tool is targeted for; the assortment of languages and technologies present in polylingual software systems, combined with the variation in developer ability make it difficult to definitively evaluate the claims laid out. Consequently, for my thesis I am interested only in dealing with

---

1 A copy of the Certification of Institutional Ethics Review letter is included in Appendix A.
an archetype of the problem of polylingual dependency analysis; this allows an evaluation of whether the technique is viable in at least some contexts, rather than all contexts. For the evaluation, I pose the following research questions:

6.1.1 Research Question 1 (RQ1):

Can a Software Developer Successfully Configure GrammarSketch?

I am interested in seeing how a developer, with some training in the GrammarSketch tool, goes about configuring the tool for their system. I would expect that a developer interested in using GrammarSketch would be familiar with the programming languages in use in their software, and would know what syntax and semantics for each language would be important for polylingual dependency analysis. However, how difficult is it to translate such knowledge into a successful configuration of the GrammarSketch tool?

To answer this question, I am interested in how long it takes a developer to configure GrammarSketch as a measure of how difficult it is to express developer knowledge in the tool’s notation. In examining the results of the tool, I will also be interested in seeing what kinds of dependencies are missed by the developer’s configuration, as an indicator as to what syntax may be difficult to describe in GrammarSketch.

6.1.2 Research Question 2 (RQ2):

Is GrammarSketch’s Efficacy Comparable to Alternative Solutions?

I am interested in the accuracy of the GrammarSketch tool once configured by a developer with some understanding of their target software system. A key component of GrammarSketch’s design has been focused on reducing the amount and complexity of information that a developer needs to provide to configure the tool for dependency analysis; this reduction is achieved by simplifying the pattern notation, and by making assumptions as to how dependency relationships are formed. However, are these assumptions and simplifications valid in practice?
A key measure for answering this question will be determining the accuracy of each configuration of GrammarSketch. Accuracy in dependency analysis can be measured in terms of *precision* (i.e., the percentage of dependencies found that are in fact dependencies, relative to the total number of dependencies reported) and *recall* (i.e., the percentage of true dependencies reported relative to the true number of dependencies present).

### 6.2 Methodology

My evaluation of GrammarSketch involved observing how software developers configured the GrammarSketch tool to detect polylingual dependencies in an open-source system. The focus of the evaluation was in two parts: (1) determine whether they were able to configure the GrammarSketch tool to their satisfaction in a relatively brief window of time, and (2) evaluate the accuracy of the polylingual dependencies detected by each configuration to establish the tool’s effectiveness.

#### 6.2.1 Evaluation Target: The OpenBravo ERP System

I decided to conduct the evaluation on the OpenBravo ERP System, which was previously used in the Luther testbed (see Section 4.3). My choice to re-use OpenBravo was predicated on two factors:

- In Section 4.3, I described how I manually analyzed the polylingual dependencies present between two files in OpenBravo’s `org.openbravo.erpCommon.utility` package. By reusing this original data, I could determine the accuracy of each GrammarSketch configuration produced by the study participants.

- The Luther testbed had been configured with a series of island grammars of varying semantic complexity, whose accuracy was measured against the OpenBravo system. By performing the evaluation again on OpenBravo, the configurations of each participant
could be compared and contrasted with the various Luther configurations to give a sense as to the relative performance of the tool.

I did not feel it was feasible to recruit participants to use GrammarSketch on their own polylingual software systems: recruiting industrial participants for software engineering studies is difficult enough without also requiring that participants (1) be currently working on a polylingual system, and (2) be willing to allow me to analyze their propriety code. Having participants apply GrammarSketch to their own system also makes it expensive to quantify the tool’s accuracy: for Luther, it took two weeks to manually analyze the polylingual dependencies present between just two source code files; the prospect of repeating that analysis for each participant in the evaluation was not enticing. Finally, the variation in the systems that could be encountered would make it difficult to compare and contrast the tool’s performance in each situation.

However, choosing to use OpenBravo ERP for this evaluation provided a major obstacle: none of the participants would be familiar with the system or its code base. While any participant selected would have a solid understanding of the semantics of the Java, XML, and SQL languages, they would not understand the semantics behind the technology OpenBravo utilizes to facilitate polylingual dependencies. To address this problem, I decided to approximate the knowledge that the participants ideally would have of the OpenBravo system, by creating a list of semantics in each language that were necessary to recognize to detect polylingual dependencies. During the experiment, participants would be able to refer to this list as needed.

6.2.2 Participant Recruitment and Selection

I recruited participants for this evaluation through an e-mail sent to graduate students currently or formerly associated with the Laboratory for Software Modification Research (LSMR), and

2LSMR is the research lab I am currently associated with, in the Department of Computer Science at the University of Calgary.
to professional software developers with whom I am acquainted. Two professional developers responded, and two graduate students from LSMR volunteered. I also asked another graduate student from LSMR to participate in a pilot of the case study; their feedback identified early issues with the GrammarSketch tools, and confusion in some of the case study instructions, which I addressed before continuing the case study with the rest of the participants. Each of the participants was familiar with the programming languages involved in the study. Both graduate students were near the end of their degree programs with one already working as a professional developer. Both professionals had at least four years of industrial software development experience.

6.2.3 Evaluation Overview

I chose to evaluate GrammarSketch by performing a case study where participants are asked to configure the tool to recognize polylingual dependencies in the OpenBravo system.

Each participant was provided with a customized version of the Eclipse IDE that contained:

- the GrammarSketch plugin;
- a set of examples from an online tutorial I created (see Appendix D), involving the use of GrammarSketch to find simple dependencies in Java source code; and
- the .java and .xsql source code files from the org.openbravo.erpCommon.util-ity package in the OpenBravo ERP system, to be used to test their configuration against (39 source code files in total).

Participants were asked to work through a tutorial that showed how to use the GrammarSketch tool, and how to write patterns using its notation. The examples in the tutorial would be provided with the tool so that participants could see how they worked, and also experiment with the examples to further their understanding. Participants were given as much time as they felt necessary to familiarize themselves with the tool.
When each participant decided they were ready to conduct the study, they were directed to a webpage outlining the case study task: to configure GrammarSketch to recognize polylingual dependencies between Java, SQL, and XML code in the OpenBravo system. The webpage described the semantics in each programming language that need to be recognized to find polylingual dependencies in the system, and participants were only required to write as many patterns as they felt were needed to recognize the relevant syntax. Participants were allowed to spend up to one hour configuring the tool, but were allowed to stop once they felt the configuration was complete.

I then collected the grammarsketch.gsk files each participant created during the study to configure GrammarSketch. I used each of these files to configure GrammarSketch, and to analyze the polylingual dependencies between the two files in the org.openbravo.erpCommon.utility package for which I had previously determined the dependencies present: VerticalMenu.java and Menu_data.xsql. The dependencies predicted by the tool for each configuration were compared against the dependencies previously determined to be present in the system, and from this the precision and recall of the dependency analysis, accuracy was evaluated.

6.3 Results

In this section, I present the observations and results of each participant in the case study. The configurations created by each participant can be found in Appendix E.

6.3.1 Characterization of Dependencies

In Section 4.3.1, I described the procedure I followed for determining the polylingual dependencies present in the source code. In this section, I further describe the dependencies discovered to illustrate what I expect GrammarSketch to find.
The 105 polylingual dependencies between the *VerticalMenu.java* and *Menu_data-.xsql* files can be broken down as follows:

- 6 are between method invocations in Java, and XML element tags.
- 49 are between type declarations and references in Java, and SQL table names.
- 8 are caused by the change of scope that occurs when an XML element tag encapsulates a SQL query.
- 42 are between string literals in Java, and string literals in SQL.

The last two categories of dependencies are difficult for GrammarSketch to detect:

1. Currently, GrammarSketch does not track the nesting level of any identifier it finds as this information was found to be of little benefit in the Luther case study (see Section 4.4). Such changes in scope would likely be obvious to a developer, but not obvious to tool support.

2. The string literals in the OpenBravo code are used in both Java and SQL, but are combined with a level of indirection: in Java, the string literals occur in a few method bodies where they are compared against a parameter passed to that method whose type is a *String*. In SQL, the literals are compared against a parameter retrieved from a field in a database table. Resolving the dependency between these string literals is not impossible, but it is quite involved. As such, detecting these dependencies would require (1) some sophistication in the understanding of the dependencies in the OpenBravo system, and (2) specialized patterns that may have limited usefulness elsewhere in the system.

Since the participants in the case study are expected to not have a sophisticated understanding of the dependencies in the software (due to lack of familiarity with the OpenBravo source code), it is unlikely that the 42 dependencies between string literals will be detected. Further,
GrammarSketch is currently incapable of detecting the 8 dependencies caused by a change of scope. Consequently, I suggest that 55 of the 105 dependencies present in the system are reasonably detectable. This implies that a recall of 52.38% is the most that can be expected of the participant’s configurations.

6.3.2Participant 1 (P1)

P1 is a professional developer who conducted the study remotely, and was the first participant in the study. P1 has worked as a professional developer for four years, and works with the C++ language primarily.

The participant e-mailed me to notify me that they were starting the experiment; the next contact I had with them was an hour and 10 minutes later where they notified me that they could not even get the examples provided in the tutorial to work. I had no feedback at the time as to what problems they were having and could not intervene. After notifying me that they were unsuccessful so far, I asked P1 to send me what they had managed to accomplish so far in their **grammarsketch.gsk** file, which is included in Appendix E.1. A discussion of the problems P1 faced is provided in Section 6.3.3

Unsurprisingly, the **grammarsketch.gsk** file submitted by participant P1 failed to detect any polylingual dependencies in the source code. This does not mean that P1 was not able to successfully write any patterns: all but one of the patterns written by P1 were able to find matches in the source code. The reason no polylingual dependencies were detected was that four of the seven patterns written by P1 used only the *reference* syntax. GrammarSketch interprets identifiers recovered as GSK-References as having a meaning local to the file they are discovered in, and will not attempt to match that identifier outside of the file. This could have been easily fixed had P1 been able to use GrammarSketch to apply the patterns to the source code, so that P1 could have had feedback on how their patterns worked.
6.3.3 Lessons Learned From Participant 1

The failure of the first case study with P1 highlighted some important deficiencies with the case study that the pilot study had failed to reveal. Most of these issues could have been easily resolved had I been present with the participant during the case study, but nearly all of the participants were conducting the study remotely due to logistical issues. I wish to take some time to discuss this problem to both note (1) what I learned, and (2) what corrective steps were made as a result.

The problems P1 faced were due to issues with and idiosyncrasies of the Eclipse IDE that they were not familiar with:

1. On startup, the Eclipse IDE asks the user where their source code is located on their computer. When I configured the Eclipse IDE for the case study, I set the location in Eclipse to C:\Eclipse\workspace, which is the standard. I placed instructions on the website that if the location Eclipse was installed to (e.g., C:\Eclipse) was changed, the workspace location would also need to be altered. This necessary step was overlooked.

2. Eclipse maintains a separate state of the contents of the file system that can become desynchronized with the actual contents of those directories. After installing the Eclipse configuration provided for the case study, Eclipse became desynchronized with the included examples and source code files. While files and projects will appear to be present, any attempt to programmatically retrieve the contents of the file will results in a null pointer being sent; opening such files will result in a blank editor window, with a short message that the file system needs to be refreshed. To fix this, a user would need to either (1) right-click on the project and select “Refresh”, or (2) select the project and press the “F5” key. This would not likely be known to new users.

Consequently, once P1 had completed their install, they were not able to get Grammar-Sketch to work on any of the supplied code sets. When they created their own examples,
GrammarSketch seemed to work (as Eclipse was synchronized with those newly created files, but not with files provided with the IDE). The message GrammarSketch gave for that error was simply that it could not find any matches, which was not the actual underlying problem.  

3. The `grammarsketch.gsk` file is a plain text file that is compiled by the GrammarSketch plugin. Participant P1 did not realize for a considerable portion of the study that after a pattern had been written to the file, they needed to save the file before they tried to run the GrammarSketch plugin. When they tried to save the file, they used a metaphor that was appropriate for a different IDE—in the Microsoft Visual Studio IDE, one can right-click on the window tab corresponding to the `grammarsketch.gsk` file to save it to disk. Eventually, the participant figured out how to save the file using the keystroke `CTRL-S`.

Another area of serious concern was that during the experiment, I was online using an instant-messaging tool, and had asked P1 that if they had any issues or concerns they should contact me. This never happened; P1 chose to struggle with the issues they were having and did not notify me that they were having difficulty until they chose to terminate the case study. This suggested that the participant confused the difficulty they had with the Eclipse IDE and the GrammarSketch tool with a lack of skill or ability on their part, rather than associate those problems with deficiencies in the GrammarSketch tool or Eclipse IDE. A follow-up conversation confirmed the participant felt this way.

To rectify this situation, I had subsequent remote participants install a Virtual Network Connection (VNC) client called TeamViewer\(^4\). This allowed me to observe what the participant was doing and, if needed to, intervene to rectify any problems. This allowed me to mitigate

---

\(^3\)This very basic issue did not come out during the pilot study, as the graduate student participating in the pilot study had a previous installation of Eclipse. They imported just the plugin and source code into their IDE, but did not reinstall Eclipse, and consequently did not trigger this problem.

\(^4\)http://www.teamviewer.com, as of 2008/05/05
issues participants had that were caused by problems with either the GrammarSketch interface, or the Eclipse IDE.

6.3.4 Participant 2 (P2)

P2 is a professional developer who conducted the study remotely. P2 has been a professional developer for five years, and has professionally developed Java, XML, and SQL software systems for two of those years. The participant downloaded the customized version of Eclipse, and then installed the VNC client so that I could remotely observe what they did.

P2 spent just under an hour (55 minutes) familiarizing themselves with the GrammarSketch tool and the tutorial examples. Once the case study started, they spent 18 minutes configuring the GrammarSketch tool for OpenBravo at which point they were satisfied with the results it was giving them.

Dependency Detection Accuracy
P2’s grammarsketch.gsk file is included in Appendix E.2. After analysis, P2’s configuration of GrammarSketch detected a total of 292 dependencies across three languages. Of those matches, 55 were real. My previous work indicated that there were 105 polylingual dependencies present in the system; this gives the P2 configuration a precision of 18.84%, and a recall of 52.38%.

Observations
One behaviour of P2 that stood out is that they wrote one pattern at a time, refined it, and then when the pattern was complete erased the pattern, and moved on to the next pattern. They did this so that they could independently test and verify the accuracy of each pattern they wrote, rather than having problems with a single pattern be obscured by the set of other patterns they had written.

However, this also may have contributed to a high rate of inaccuracy in their results. In
the final pattern set they submitted, one pattern did not use any type or reference syntax: the pattern consequently does not recover any identifiers from the source code. Another Java pattern they wrote consisted solely of type. This would erroneously capture every identifier in Java source code, and mark them as being GSK-Types. Both of these problems would have been apparent when using the tool, and in developing the patterns. Because P2 developed each pattern separately, they needed to collate the patterns they wrote into the final grammar-sketchnsk file that was submitted; these patterns were likely included by accident.

6.3.5 Participant 3 (P3)

P3 is a graduate student who conducted the study remotely. The participant regularly uses Eclipse in their work, so had no problems setting up the case study environment, and also installed the VNC client.

P3 spent 50 minutes familiarizing themselves with the GrammarSketch tool and the tutorial examples. During the period of time they performed the case study, the Eclipse IDE crashed on them twice. P3 commented that this (the software crashing) was not unusual for them, and attributed the problem to issues they had with their computer. To compensate for the time lost, I extended the study duration an additional 20 minutes to 80 minutes. P3 took 77 minutes to complete the configuration of GrammarSketch.

**Dependency Detection Accuracy**

P3’s grammarsketch.gsk file is included in Appendix E.3. After analysis, P3’s configuration of GrammarSketch detected a total of 144 dependencies across three languages. Of those matches, 30 were real. This gave their configuration a precision of 20.83%, and a recall of 28.57%.

P3’s patterns in the GrammarSketch missed all 6 dependencies between method invocations in Java and XML element tags. The problem stemmed from one pattern: the pattern XML:NameValuePair in Appendix E.3 correctly defines how to recognize the “name” at-
tribute that indicates that this XML tag element is mapped to an Java method invocation; however, P3 recovered the identifier associated with “name” using STRING_reference. P3 should have used STRING_type.

P3 also missed 19 of the 46 dependencies between Java types and SQL tables. P3 did not take into account types and references occurring in the context of array declarations and accesses, which adds a slight variation to the Java syntax. But this may in part be due to relatively uncommon nature of array declarations in the code: only 13 of the 23 Java files provided for the participants from OpenBravo used arrays. It is likely that P3 tested their patterns on portions of the code that did not have array declarations, and so did not notice this oversight.

Observations

P3 had several problems with numerous grammarsketch.gsk windows being open, and editing the wrong files. To help participants prepare for the study, I created four projects containing the source code and GrammarSketch patterns used in the tutorials (there were four tutorials). Participants could see how GrammarSketch worked, and could also modify the examples and/or patterns to better understand the behaviour of the tool. During the case study, participant P3 left open several grammarsketch.gsk files from differing projects, and as such would get confused as to which window was relevant for the current project. They would then modify the wrong /gsk/grammarsketch.gsk file, and be confused when GrammarSketch would not detect any new matches.

P3 also was confused by the semantics needing to be recognized in the XML source code. OpenBravo uses a specific XML attribute value to indicate that a mapping exists between a Java method and that particular XML element. The developer spent nearly 20 minutes at the end of their case study, trying to refine a pattern meant to recognize all attributes of all XML elements, but was not seeing the results that they wanted: the problem was they were
recognizing all XML attributes, but not the value associated with the specific XML attribute that was relevant. They were able to eventually fix their mistake.

6.3.6 Participant 4 (P4)

P4 was a graduate student who conducted the study at the Laboratory for Software Modification Research (LSMR) offices. The participant regularly uses Eclipse in their work, and had no problems setting up the case study environment. Because the study was conducted within the research lab, there was no need to set up the VNC client.

P4 spent 30 minutes familiarizing themselves with the GrammarSketch tool, and the tutorial examples. Once the case study started, they spent 22 minutes configuring the GrammarSketch tool. Participant P4 had very few of the problems the other participants had, partly due to their familiarity with the Eclipse IDE tool.

Dependency Detection Accuracy

P4’s $grammar\text{\_sketch}.\text{gsk}$ file is included in Appendix E.4. After analysis, P4’s configuration of GrammarSketch detected a total of 149 dependencies across three languages. Of those matches, 37 were real. This gave the P4 configuration a precision of 21.48%, and a recall of 30.48%.

P4 missed 23 of the 46 dependencies between Java types and SQL Tables. Most of the dependencies missed were for the same reason as P3—failure to account for array syntax in Java. P4 also encountered an unusual problem: they wrote two patterns, one for SQL and one for XML, that used reference syntax, when type may have been more appropriate. As such, the identifiers recovered by these patterns were treated as having meaning local to the SQL/XML file in which they were found. However, OpenBravo combines XML and SQL within the same file, so GrammarSketch resolved the dependencies properly within those files, detected polylingual dependencies, and highlighted the developer’s code appropriately with purple; the developer had feedback that they had successfully found polylingual dependencies,
and so stopped modifying the pattern. Had the developer used the `type` syntax instead, they would have found additional polylingual dependencies in the Java source code, and improved the recall of their configuration.

**Observations**

P4 was prolific in writing GrammarSketch patterns, creating 13 patterns in the course of their work (P3 had the second most at 8). A number of the patterns are variations on each other, each restricting the cases in which the patterns could be matched. However two of their patterns had absolutely no effect on dependency analysis, as they do contain any `type` or `reference` elements. As a result, nothing is extracted from these patterns for dependency analysis. If P4 had developed each pattern separately as P2 did, they may have seen this problem and either modified the patterns, or removed them.

6.3.7 Expert Configuration

To provide a comparative contrast with the results of each participant, I have also provided an “expert” configuration of GrammarSketch, which is the configuration I have created for recognizing the same polylingual dependencies in OpenBravo. The configuration took approximately 25 minutes to create.

**Dependency Detection Accuracy**

The expert configuration is included in Appendix E.5. After analysis, the expert configuration detected a total of 149 dependencies across three languages. Of those matches, 55 were real. This gave the expert configuration a precision of 36.91%, and a recall of 52.38%.

6.4 Implications of Results

In this section, I discuss the observations and results obtained from the case study outlined in Sections 6.2 and 6.3. In Section 6.4.1, I discuss the observations and results relevant to research
question RQ1, and in Section 6.4.2 those that are relevant to question RQ2. In each section, I present a summary of the results and observations, the conclusion I draw from the result, and what significance these results present for the GrammarSketch tool.

6.4.1 RQ1: Can a Software Developer Successfully Configure GrammarSketch?

All of the participants were able to write patterns to recognize syntax in each of the languages in OpenBravo, including participant P1 who could not get the GrammarSketch tool to function properly. Participants P2, P3, and P4 were able to configure GrammarSketch to recognize polylingual dependencies, in under an hour and with limited training in the tool. Two of the participants were able to configure the tool in roughly 20 minutes, which speaks well to the simplicity of the notation used by GrammarSketch.

However, a few errors consistently appeared during the study, suggesting the configuration of GrammarSketch can be improved. Participants generally did not test their patterns in isolation, instead choosing to add additional patterns to those they had already written. This contributed to participants writing patterns that had limited or no effect in recovering dependencies from the code. In the case of P1 who developed their patterns in isolation, the collation of the final set of patterns proved to be problematic as well.

Another problem encountered was confusion between the applicability of the type syntax over the reference syntax. Participants P3 and P4 both wrote patterns where no type element was present, only reference elements, making those patterns suitable only for matching dependencies within the file. It is not clear if the confusion here is conceptual, poor feedback from GrammarSketch, or both. Participants writing patterns using only reference elements would see yellow highlighting indicating where GrammarSketch had matched their pattern to the code, and might have interpreted the highlighting as their pattern being successful. However, none of the highlights would have been the purple used by GrammarSketch to indicate that it had detected a polylingual dependency. Participants would likely not have had enough
experience to recognize the absence of purple highlighting as being important feedback on their patterns.

**Conclusions**

Participants were able to configure the GrammarSketch tool, but their success is currently limited by poor feedback mechanisms in the tool, preventing them from easily ascertaining which of their patterns are ineffective.

**Implications for Tool Support**

Much of the difficulty participants had could have been alleviated by providing support within GrammarSketch to write and test patterns in isolation, adding them to their `/gsk/grammar-sketch.gsk` configuration file when they were satisfied. This would have addressed issues with ineffective patterns being written by participants P3 and P4, as well as the inclusion of erroneous “practice patterns” by P2 due to human error in reassembling the patterns developed in isolation.

The second area of concern is the potential for conceptual confusion between the appropriateness of `type` and `reference` syntax. While better training of the participants prior to the case study could have mitigated this, a more effective approach would be to improve the feedback on patterns matched by GrammarSketch, such that the user could recognize when they have used inappropriate syntax. Currently, the feedback GrammarSketch provides is primarily in the form of colour highlighting of the source code; users can see the dependencies on a given identifier, but only after deliberately selecting it. Another means of gauging pattern effectiveness is needed.

6.4.2 RQ2: Is GrammarSketch’s Accuracy Comparable to Alternative Solutions?

In Section 4.3, I determined the accuracy of several configurations of the Luther testbed on the same set of files used in the evaluation described in Section 6.2. Each configuration was
meant to approximate a level of semantic awareness of the Java, SQL, and XML languages. By comparing the precision and recall of each GrammarSketch configuration against the accuracy of those Luther configurations, I may get a comparative sense of how effective that participant’s configuration of GrammarSketch is. The precision and recall information for each Luther and GrammarSketch configuration are found in Tables 6.1, and 4.1 respectively.

The precision P2, P3, and P4’s configuration is significantly better than any of the lexically oriented Luther configurations (lexical, keyword, and nesting), although it is far lower than the precision of the references configuration. In terms of recall, the performance is almost inverted; the configurations of P3 and P4 have slightly worse recall than the lowest recall of the Luther configurations: references. However, we do not know what level of precision and recall is necessary for approximate dependency analysis support to be useful; a simple averaging of precision and recall is not a useful measure in this case to indicate whether one approach is better than another. For example, the references configuration of Luther in Table 4.1 reports precision and recall as 100% and 36.6% respectively. The “expert” configuration of GrammarSketch reported in Section 6.3.7 is 36.91% and 52.38%, by comparison. Which of these is better? Is being right all the time, but only recovering a third of the real dependencies in a system better than having to filter out two-thirds of your results but at least being able to recover half of all the real dependencies?

Luther also evaluated each configuration in terms of the effort required to create that configuration; the effort required to configure the Luther testbed ranged from 630 lines-of-code (LOC) for the simplest configuration, to 1028 LOC for the references grammar (see Table 4.1). By comparison, the configuration files submitted by each participants for GrammarSketch range in size from 23 to 51 LOC. This is not an entirely fair comparison, as the Luther testbed was built on top of the JavaCC parser generator, which uses a robust notation intended for specifying a complete language grammar. Further, LOC is not an entirely apt measure for effort, as discussed in Section 4.5. That being said, the Luther testbed configurations were generated
by someone with significant expertise with both the JavaCC parser generator, and the island grammars technique, and still took a significant amount of time to create. By comparison, the GrammarSketch configurations of each participant reflect the effort of someone with limited exposure to the tool, little familiarity with the source code being analyzed, and under an hour’s worth of effort in creating the configurations. The order-of-magnitude difference in the size of the configuration files needed for GrammarSketch and Luther is suggestive as to how much simpler this approach is compared to more sophisticated dependency analysis techniques like island grammars.

**Conclusions**

GrammarSketch is cheaper to configure than alternative approaches, but the results do not yet show that its accuracy is comparable to, or significantly better than the alternatives, partly because it is not clear as to what levels of precision and recall are necessary for approximate dependency analysis to be useful. For now, RQ2 remains a partially open question.

**Implications for Tool Support**

Unfortunately, the results of the case study do not provide clear guidance as to why the GrammarSketch configurations of each participant are less effective than the “expert” configuration. The case study preparation did not provide any testing to demonstrate that each participant was adequately trained to use the GrammarSketch tool prior to undertaking the case study. Substantial improvements to the accuracy of each configuration could be made with additional experience and training with the tool. Considering that the participant’s configurations, despite limited exposure to the tool, are not so far off the “expert” configuration, it is highly probably that adequate training prior to conducting the study would result in accuracies much closer to the “expert” configuration.
<table>
<thead>
<tr>
<th>Participant ID</th>
<th>Dependencies Detected</th>
<th>True +’s</th>
<th>False +’s</th>
<th>False −’s</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>P2</td>
<td>292</td>
<td>55</td>
<td>237</td>
<td>50</td>
<td>18.84</td>
<td>52.38</td>
</tr>
<tr>
<td>P3</td>
<td>144</td>
<td>30</td>
<td>114</td>
<td>75</td>
<td>20.83</td>
<td>28.57</td>
</tr>
<tr>
<td>P4</td>
<td>149</td>
<td>37</td>
<td>112</td>
<td>68</td>
<td>21.48</td>
<td>30.48</td>
</tr>
<tr>
<td>expert</td>
<td>149</td>
<td>55</td>
<td>94</td>
<td>50</td>
<td>36.91</td>
<td>52.38</td>
</tr>
</tbody>
</table>

Table 6.1: GrammarSketch Detection of Polylingual Dependencies

6.5 Summary

The GrammarSketch tool was evaluated using a case study on the OpenBravo system. Four participants configured the GrammarSketch tool to detect polylingual dependencies for OpenBravo, and the accuracy of each configuration was compared against two source code files for which the actual dependencies present were known in advance. Table 6.1 summarizes the results of the case study conducted. One of the participants was not able to complete the study. The remaining three participants were all able to configure the tool in the time allotted, and with roughly equivalent precision. The recall accuracy of each participant varied the most.

The results of the case study provide some support for the claims of my thesis. In addressing Research Question 1, the case study results do suggest that developers are able to successfully configure GrammarSketch for polylingual dependency analysis, but the tool’s weak support for providing feedback on pattern effectiveness leads to confusion in some cases that in turn limited the participant’s success. However, in addressing Research Question 2, the case study is only able to suggest that GrammarSketch is likely cheaper to configure than alternative approaches; the results do not clearly indicate whether GrammarSketch is comparable to alternative approaches in its accuracy. The results of the case study do suggest specific areas in which GrammarSketch can be improved, which in turn should also provide developers with more support in effectively configuring the tool for their software systems.
Chapter 7

Discussion and Future Work

In this chapter, I present a critique of the case study described in Chapter 6, and outline what other areas of research I am interested in pursuing with GrammarSketch. In Section 7.1, I critique the validity of the study conducted, by discussing sources of error in the results and what limitations must be placed on the conclusions drawn from these results. In Section 7.2, I outline my plans for improving GrammarSketch as a tool, other avenues of research that may contribute to improving this approach, and what future evaluations of this approach are planned.

7.1 Limitations and Sources of Error

The case study I conducted has several limitations that affect the conclusions I may draw from this data.

7.1.1 Critique of Construct Validity

In examining the construct validity of the case study, I am asking whether the case study as conducted in fact answers the two research questions set forward in Section 6.1. There are two confounding factors caused by the case study design:

**Participant’s training with the GrammarSketch tool varied heavily**

As discussed in Section 6.4.1, participants were not tested as to their understanding of the GrammarSketch tool prior to conducting the case study. Measurements of the GrammarSketch tool accuracy are consequently obscured by not understanding whether the developer using the tool understands what they are intending, or is still learning as they perform the study.
Comparing the participant’s results with the “expert” configuration provided in Table 6.1, it would seem that the accuracy reported is more reflective of what each participant learned in the limited exposure they had with the GrammarSketch tool, rather than the overall effectiveness of the GrammarSketch tool.

**Participant’s were unqualified to judge when the tool was correctly configured**

Participants were given the option to end the case study when they felt that had sufficiently configured GrammarSketch for OpenBravo. Since participants were unfamiliar with OpenBravo, and relied on a instruction list to describe the polylingual dependencies present in the source code, participants were not really qualified make that judgement call. I expected that participants would methodically write patterns to address the list of dependency semantics I had provided, and use this to inform them as to whether they had fully configured GrammarSketch. The lower than ideal recall values suggest this may not have been a reasonable assumption to make.

7.1.2 Critique of Internal Validity

In examining the internal validity of study, I am interested in examining what factors present in the case study could interfere with the results and the observations made.

**Presence of example code**

During the study, all participants were allowed to refer back to the tutorial and to the examples provided in the pre-configured Eclipse IDE. Participants may have relied heavily on the examples to guide their configuration of the tool, reforming and adapting the examples in some cases rather than generating their own patterns.

However, it would be unreasonable to expect that a developer would need to memorize and configure the GrammarSketch tool without the assistance of any documentation or resources. Further, the intended use case of the tool is that it be configured by a single developer, and
reused by other developers on the team. After its initial configuration, I would expect that subsequent modifications to the configuration would then be infrequent and perhaps not by the original developer. In such a case, the modifier would likely be unfamiliar with the tool and would need to rely heavily on examples and tutorials to learn how to make the desired modification.

**Human error in determining polylingual dependencies**

The accuracy of each configuration is measured by comparing predictions against a pre-determined set of dependencies discovered through manual analysis of the OpenBravo source code, described in Section 4.3.1. It is very possible that the set of “correct” dependencies I determined in this way is not complete: I may have missed dependencies, or wrongly interpreted a dependency relationship existing where in fact none did. I am not a developer on the OpenBravo system, nor did I conduct this analysis with feedback from anyone on the OpenBravo team.

However, I have previously worked on similar polylingual systems that use various data-mapping techniques similar to those employed in OpenBravo to allow Java, SQL, and XML code to interact. Further, in cases where I was not sure if the semantics implied a dependency existed in the two files examined, I was able to look for similar contexts elsewhere in OpenBravo to see if the syntax was consistent or different. In cases where it was still not clear as to whether or not a dependency existed, I chose to ignore the presence of a potential dependency. As a result, I may have under-reported the number of actual dependencies present in the system. However, the purpose of this work is a comparative analysis of the accuracy of each configuration, rather than the definitive statement of what the accuracy is: the consistent treatment and measurement of these dependencies should mean that any relative comparison will still be valid.
7.1.3 Critique of External Validity

In considering in what way I can generalize the results of the GrammarSketch case study to dependency analysis on polylingual systems in general, I must carefully consider the following factors:

**Lack of a concrete change task in the case study**

Participants were asked to only configure the GrammarSketch tool for the case study; they were not required to use GrammarSketch as part of an investigation for a change task, or to assist them in enacting a change. In doing so, I have not directly evaluated whether and how GrammarSketch aids developers in evolving polylingual software systems. I note though that the work of other researchers in the context of developing dependency analysis support follows a similar approach. In Sections 3.3, 3.4, and 3.5, eight papers are presented from researchers applying varying approaches to creating tool support appropriate for dependency analysis. Of those eight, only one paper uses their proposed technique to actually assist the enactment of changes to a proposed software system [Moo02]. Other researchers felt it sufficient to demonstrate that their tool was able to recognize dependencies in the software system; few, if any, attempted to qualify their technique’s accuracy. By comparison, the effectiveness of each configuration of GrammarSketch in the case study was quantified, and contrasted against previous studies on the same system to provide a comparative analysis of the tool’s effectiveness. I suggest this is a sufficient basis for considering the tool’s appropriateness for investigating dependencies in polylingual source code.

**Uniqueness of OpenBravo**

OpenBravo is only one example of a polylingual software system; the JPetStore system described in Section 2.1 also uses Java, SQL, and XML code, but uses a different technology

---

1. It should be noted there are unique reasons for this: Moonen’s research paper presents work he performed under contract for a Dutch bank [Moo02]. Undoubtedly his employers would have been unsatisfied had he only demonstrated that his approach worked for the changes they wanted.
to facilitate interaction. Other such systems may use languages and technologies for which the approach used in GrammarSketch may be wholly unsuitable. There are three limitations GrammarSketch currently has that would hinder its use for other systems.

1. The tool relies on a C-style syntax to be present in the source code, due to assumptions it makes on what constitutes an identifier. It is not clear at the moment where this may cause a problem.

2. GrammarSketch is not well suited for languages where the use of whitespace (e.g., indentation, carriage returns) is part of the language’s syntax. The two more notable languages that use such syntax (Ruby and Python) are also dynamically typed languages, for which the analysis that GrammarSketch attempts is largely inappropriate.

3. The tool also does not currently support the aliasing of an identifier by applying a prefix or a simple lexical transformation, such as is currently used by JNI (see Sections 3.4.1 and 4.5).

My assumptions as to the suitability of using dependencies, types, and references, as the basis of dependency analysis for other polylingual technologies may also be unsuitable. Without a means of conceptually comparing how technologies facilitate dependencies, or a map of concepts common to various programming languages, it is difficult to extrapolate for what kinds of software systems GrammarSketch is appropriate.

7.2 Future Work

My current work with the GrammarSketch tool is the culmination of several goals outlined in Section 4.5.2. Several of the participants during the case study commented that they found the tool interesting to use, and could see applicability to their own work. The road ahead for Gram-
marSketch focuses on two areas: addressing shortcomings in the tool itself, and expanding on the studies conducted so far.

### 7.2.1 Short-Term Improvements to GrammarSketch

There are several immediate improvements that can be made to the GrammarSketch tool, primarily to improve the usability of the tool. In Sections 6.4.1, 6.4.2, and 7.1.3, I outlined the following improvements for GrammarSketch based on conclusions drawn from the case study:

1. explicitly support the development and testing of GrammarSketch patterns in isolation from each other, storing the patterns into a master file when the user is satisfied;

2. provide richer feedback mechanisms to indicate where and how patterns are matching the source code;

3. explore what other kinds of information should be captured by the GrammarSketch pattern notation to improve accuracy; and,

4. allow the definition of alias rules to support languages and technologies where a dependency may be formed between lexically different identifiers.

### 7.2.2 Future Research in Polylingual Dependency Analysis

I am interested in trying to increase the sophistication of how GrammarSketch interprets patterns and resolves dependencies between pattern matches. Two of the participants wrote patterns to match Java code that failed to catch a special case (array syntax) that is not all that different from the pattern they wrote. One of the principles of GrammarSketch’s design is that it should eliminate details that hinder the expression of intent: ideally, the tool should avoid requiring developers to write multiple variations of patterns that are conceptually the same, but are intended to capture a variety of special cases with slightly altered syntax. Instead, I would like to explore techniques in which patterns can be alternatively applied as a heuristic of
what should be matched; matches for such patterns would have varying thresholds applied to
determine if a particular string is accepted, rather than using a strict “accept/reject” approach.

I would also like to explore ways in which the assumptions made by GrammarSketch in
resolving dependency relationships for the developer, can by augmented with machine learning
techniques. In using the tool, especially across a software development team, developers will
frequently encounter cases where a dependency detected by GrammarSketch is clearly wrong. I
am interested in adding support for developers to provide feedback that a particular dependency
is correct or incorrect, and then leverage that feedback in some way so that GrammarSketch
can refine its dependency model to more closely mimic how that particular system behaves.

7.2.3 Future Evaluations of GrammarSketch

In Section 5.5, I outlined the workflow for which GrammarSketch is intended. My thesis
research does not address whether GrammarSketch proves to be helpful during the investigation
in actually affecting the change to the software system (see Section 7.1.3). Consequently, my
immediate research focus is in evaluating this last concern: does GrammarSketch prove helpful
to developers who are making modifications to polylingual software system?

I am currently working on establishing a modification task, most likely on the OpenBravo
ERP system, that requires a participant trace polylingual dependencies in the source code to
investigate a proposed change task. The participant would either configure the tool themselves,
or be provided with a pre-configured version of the GrammarSketch tool that they could modify
further if they wished. The study would evaluate how the participant used the GrammarSketch
tool in their investigation, how the participant dealt with the false positive dependency matches
returned by the tool, how they dealt with the absence of dependencies reported by the tool
that may or may not affect their modification task, and whether the participant was ultimately
successful in investigating and potentially enacting the change task.

Long term research goals would involve releasing GrammarSketch to potentially interested
users, and observing their usage of GrammarSketch in their own work on disparate polylingual systems with differing team members. Such a long term research goal could be facilitated by instrumenting the GrammarSketch tool log the activity of users who consent to allow us to monitor their usage of the tool.

7.3 Summary

The results of the case study presented in Chapter 6 have several limitations which must be kept in mind, when interpreting these results. The case study has weak construct validity, as the experimental setup does not ensure that each developer is fully capable of using the GrammarSketch notation and tool. Thus the accuracy results reported for GrammarSketch commingle an evaluation of the tool’s effectiveness in dependency analysis, with the efforts by participants to teach themselves how to use the tool with imperfect understanding. The case study has a stronger internal validity, as the presence of example code and the potential for human error in determining real dependencies present in the OpenBravo source code are not appreciably influencing factors. The external validity is currently limited to those systems which share similar languages and technologies to those used in OpenBravo.

The case study results suggest several concrete improvements which can be made to GrammarSketch to address many of problems participants encountered during the case study. Future research efforts with the approach used by GrammarSketch will focus on augmenting the notation scheme used to support additional syntactic patterns, techniques which can infer syntactic variations on patterns for developers, and further evaluations which try to attempt questions left unanswered by this case study.
Chapter 8

Conclusion

Dependency analysis is one of the techniques relied on by software developers, who are responsible for maintaining and evolving software systems. As software systems have increased in size and complexity, research into new techniques and tools have provided developers with the means to augment their own ability to investigate dependency relationships in code, and ascertain what the impact of changes to their software systems will be. This support is lagging now; software developers are taking advantage of technologies that allow them to create software systems using multiple programming languages. Tool support that is effective in a monolingual software system, becomes ineffective in a polylingual software context, and developers have fewer resources to leverage in maintaining these polylingual systems.

Other researchers have investigated approaches that either directly address the problem of providing polylingual dependency support, or are adaptable to this end. Lexical analysis techniques are lightweight enough that they can be retargeted for the specific combination of languages in such systems by a single developer, but require significant skill to configure, and suffer from imprecision. Syntactic analysis techniques by contrast are powerful, precise, and easy to use by developers, but are so heavyweight to construct that few can afford to build such support for their systems, and rely on adapting (if available) existing syntactic tools for their needs.

A hybrid approach that uses island grammars to provide dependency analysis support, seemed to offer a unique approach to this problem; island grammars allow the partial specification of syntactic information for a programming language, allowing better-than-lexical precision in analysis while being less expensive to create than full syntactic analysis support. To explore the suitability of this approach for polylingual dependency analysis support, I created
the Luther testbed to evaluate the accuracy and development cost of several island grammars for analyzing the dependencies in the OpenBravo system. The case study I conducted suggested that *declaration*, *type*, and *reference* semantics may be better to recognize for conducting polylingual dependency analysis, but still pointed to problems with the cost of developing island grammars for such analysis.

Out of this work, I developed my thesis: that lightweight polylingual dependency analysis tool support can be provided by imposing on the developer which semantics must be recognized for dependency analysis, and using the developer-specified syntactic patterns for those semantics to generate approximate tool support comparable in effectiveness to lexical analysis alternatives. This thesis led to creation of a research prototype tool, called GrammarSketch. GrammarSketch provides a simplified notation scheme for developers to specify the syntax patterns for those pre-selected semantics used for dependency analysis, and assists developers in identifying and exploring the dependencies in their source code. While I chose to base GrammarSketch on lexical analysis techniques due to technical difficulties with island grammars, GrammarSketch aims to overcomes the problems associated with writing complex regular expression patterns by incorporating island grammar concepts into the notation scheme used for the tool. Developers are able to express semantic patterns simply in GrammarSketch’s notation,
which the tool then transformed into more sophisticated regular expressions for polylingual dependency analysis.

To evaluate the claims of my thesis, I conducted a case study in which participants were asked to configure GrammarSketch, to recognize the polylingual dependencies present in a portion of source code from the OpenBravo ERP software system. Table 8.1 presents the results of the case study; the results of the Luther case study are also provided as contrast. These results suggest that developers are able to successfully configure GrammarSketch to provide polylingual dependency analysis support, but do not give a clear basis to say that the effectiveness of such support is comparable or better than alternative approaches; the feedback mechanisms in the GrammarSketch tool currently do not provide sufficient information to its users on flaws in their patterns, such that developers can easily identify where refinements should be made. The case study results do suggest concrete steps that can be taken to address this shortcoming.

Participants in the case study were interested in the GrammarSketch tool, and the case study results do suggest that this approach is promising. The focus of future work with the GrammarSketch tool will be on:

- addressing limitations in GrammarSketch highlighted by the case study;
- exploring additional notation to improve developer refinement of their syntax patterns;
- attempting to infer syntactic variations on developer patterns, to avoid the need to specify special cases for otherwise conceptually similar semantics; and,
- conduct an evaluation of how developers use GrammarSketch in an actual change task investigation.
8.1 Contributions

The novel contributions of this thesis are:

1. a cost-to-accuracy evaluation of island grammars developed for dependency analysis in polylingual software systems;

2. a lightweight technique for providing polylingual dependency analysis support, generated through presuming which semantics, based on empirical evaluation, are most effective for such support, and using developer-specified syntactic patterns as the sole input to recognize such semantics, and configure approximate dependency analysis tool support; and,

3. an evaluation of the GrammarSketch tool’s ease of configuration, and effectiveness in dependency analysis support.
Bibliography


Appendix A

Certification of Ethics Approval

Figure A.1: *Certification of Institutional Ethics Review* Letter from the Conjoint Faculties Research Ethics Board for the University of Calgary.
Appendix B

Division of Labour with Co-Author

As per University of Calgary thesis guidelines, all work published with other authors that is included in this thesis must be accompanied by a short description of the division of labour in creating the publication.

The work included in Chapter 4 was previously published with my supervisor as my co-author [CW07]. The research presented in the paper was developed, implemented, and evaluated by myself, with my supervisor providing guidance, feedback, and suggested directions for my work. In writing the paper, I was the principle writer of the work, with my supervisor acting in an editorial role, to help me present my ideas in the language and manner required for them to be acceptable to research conferences in my field.
Appendix C

Supplement for The Luther Testbed

Due to the limited space available in our publication venue [CW07], we could not provide as much detail as we would’ve liked on the grammars written to evaluate the cost-to-accuracy benefit of island grammars. Our reviewers indicated areas in which some additional detail would be helpful; we have endeavored to provide those details in this space.¹

C.1 Defining Token Attributes

To supplement our description of the attributes we defined for each token recognized by a grammar, we provide a concrete example from the JPetStore application. The attributes as defined reflect the semantic context retrieved by the References Java grammar. In cases where the grammar is less complete (e.g. Lexical, Keyword, and Nesting grammars), some attributes will have indeterminate values. Code

The following code snippet is a method declaration in the AccountSqlMapDao class, located in the com.ibatis.jpetstore.persistence.sqlmapdao package in the JPetStore 5.0 system. The identifier for the method declaration is in bold, and is the basis of the token attributes shown later.

```java
public Account getAccount(String username, String password) .
```

C.2 Defining Dependency Rules

Dependency rules are currently hard-coded into the Luther testbed. We presented an example of the reasoning we used to define rules in the paper, here we present a more formal example.

¹The contents of Appendix C were originally published at http://lsmr.cs.ucalgary.ca/projects/tre/polylingual-supplement
of the rules used to detect the presence of a dependency. Currently these rules are simply expressed as Java code.

Three rules are presented:

1. Match rule which is used by the lexical, keyword, and nesting grammars to match two tokens.

2. Type Inference on Identifier rule, used by the Java references grammar to infer if this identifier has a particular type associated with it.

3. Match invocation to declaration rule, which is used by the Java references grammar to see if a particular method invocation can be matched to its declaration elsewhere.

Dependency rules

1. Match

Given Token A, Token B, if: A.Identifier = B.Identifier

Then the tokens match Note that this rule in practice has an additional check made: since SQL is not case-sensitive, if either of the tokens has their language attribute set to SQL, the comparison is case-insensitive.

2. Type Inference on Identifier
Given Token A, if there exists Token B such that A.context = object B.context = declaration A.name = B.name

Then A.type = B.type

This rule is used by the Symbol table after parsing is complete to resolve the type information for identifiers discovered.

3. Match invocation to declaration

Given Token A, Token B, if: A.Name = B.Name A.Parent = B.Parent.Type A.Context = method declaration B.Context = method invocation

Then B is a reference to the declaration at A

Note the imprecision in this rule: it does not take into account static method invocations, local or inherited method invocations, or scope/namespace resolution. This imprecision however allows us to develop island grammars and relationships rules faster then it would take to correctly account for all of these factors

C.3 Methodology and Partial Grammar Sets

In this section, we wanted to provide some additional information as to what distinguished the grammars from each other.

C.3.1 What Constituted a Dependency

Here we try to elaborate a bit more over what was and wasnt a dependency. Its easiest to talk about what wasnt matched as a dependency:

False Positive

False Positives were dependency matches which were incorrect i.e. the tool said a dependency existed, in actuality no such dependency is present. False Positives are almost universally in
<table>
<thead>
<tr>
<th>Grammar</th>
<th>Language</th>
<th>Description of Grammar Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical</td>
<td>Java</td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All words accepted (i.e. non-word tokens ignored)</td>
</tr>
<tr>
<td>SQL</td>
<td></td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All words accepted (i.e. non-word tokens ignored)</td>
</tr>
<tr>
<td>XML</td>
<td></td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All words accepted (i.e. non-word tokens ignored)</td>
</tr>
<tr>
<td>Keyword</td>
<td>Java</td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java keyword list defined, as per JavaCC Java 1.5 grammar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All non-keywords accepted</td>
</tr>
<tr>
<td>SQL</td>
<td></td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PL/SQL keyword list defined, as per JavaCC PL/SQL grammar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All non-keyword accepted</td>
</tr>
<tr>
<td>XML</td>
<td></td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All words accepted (i.e. non-word tokens ignored)</td>
</tr>
<tr>
<td>Nesting</td>
<td>Java</td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Java keyword list defined, as per JavaCC Java 1.5 grammar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nesting level of tokens tracked based on { } ( ) tokens</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All non-keyword accepted</td>
</tr>
<tr>
<td>SQL</td>
<td></td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PL/SQL keyword list defined, as per JavaCC PL/SQL grammar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All non-keyword accepted</td>
</tr>
<tr>
<td>XML</td>
<td></td>
<td>C-Style token definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nesting level tracked based on &lt; /&gt; tokens. A production was added to ignore meta-tags (e.g. &lt;? &gt;) as these do not follow the normal nesting rules, and can affect the grammars. The contents of meta-tags were still parsed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All words accepted (i.e. non-word tokens ignored)</td>
</tr>
</tbody>
</table>

Table C.1: Description of the improvements made between the *lexical*, *keyword*, and *nesting* grammars.
<table>
<thead>
<tr>
<th>Grammar</th>
<th>Language</th>
<th>Description of Grammar Contents</th>
</tr>
</thead>
</table>
| References | Java | C-Style token definitions  
Java keyword list defined, as per JavaCC Java 1.5 grammar  
Nesting level of tokens tracked based on `{ }`  
( ) tokens  
Recognizes syntax patterns for class declarations, method declarations, type declarations, field declarations, method invocations, field and/or variable references. Ignores syntax for interface and superclass declarations (i.e. implements, extends), ignores syntax for exceptions (i.e. throws), ignores primitive return types, visibility modifiers. All non-keyword accepted |
| SQL | C-Style token definitions  
PL/SQL keyword list defined, as per JavaCC PL/SQL grammar  
Recognizes syntax patterns of table/column references, aliases of table/column names. All non-keyword accepted |
| XML | C-Style token definitions  
Nesting level tracked based on `< />` tokens. A production was added to ignore meta-tags (e.g. `<? ») as these do not follow the normal nesting rules, and can affect the grammars. The contents of meta-tags were still parsed.  
Recognizes format of XML element i.e. element name, attribute lists, contents. All words accepted (i.e. non-word tokens ignored) |

Table C.2: Description of the improvements made to the *references* grammar.
our case study the result of a lexical match between two tokens, but:

- They are semantically not the same. This would occur when two words were accidentally the same, but whose semantic contexts indicated they were not the same.

- They are not a polylingual dependency. We ignored the cases where a dependency existed, but it was not polylingual. This was a necessary restriction for two reasons: (1) to lower the cognitive burden on our part in determining whether something was or was not a dependency, and (2) to not confuse the grammar’s effectiveness in a polylingual context with its effectiveness in a single-language environment. The number of polylingual dependencies in our case was likely far less than the number of single-language dependencies which could be identified.

**False Negative**

False Negatives were dependency matches which failed to be recovered by the tool i.e. they do not appear in the results set at all. False Negatives were caused by two situations:

1. Non-lexical dependencies. In OpenBravo, the XML-SQL file had a non-lexical dependency between an XML element and an SQL query. Each query is wrapped in a single XML element which has no attributes or no nested children. As a result, by detecting the change in nesting in combination with the specific name of the XML element, its possible to know that the XML element is dependent on the enclosed SQL query. This may seem like an obvious result from a human perspective - the dependency is implicit. However, from a tool support perspective this is a non-obvious relationship. The tool needs to know that the enclosing XML element forms a dependency on the SQL query, to set up a dependency chain from the XML element identifying the query to the actual SQL query.
2. Rejecting dependencies due to insufficient contextual information. The references grammar rejected lexical dependency matches in cases where it did not have sufficient contextual information to know if they formed a dependency. In most cases, for a polylingual dependency to exist the symbol in each language on which the dependency exists needs to be the same lexically. So by simply accepting all lexical matches as dependencies, we can fairly easily make sure we detect all dependencies that exist (and of course, many false ones). To improve precision, it is necessary to step beyond a simple lexical match and start looking for other clues that would tell us to accept or reject the dependency. In our case study, understanding how a type, field, and method invocation mapped to an appropriate SQL query helped to eliminate spurious dependencies. However, this also meant we started to reject dependencies for which there was a lexical match, but not this other contextual information (e.g. dependencies on a string literal).
Appendix D

Tutorial Material for Case Study

In this Appendix, I present the materials provided to the participants of the case study outlined in Chapter 6.¹

D.1 The GrammarSketch Notation

In your project directory, there’s a file called /gsk/grammarsketch.gsk. This is a text file in which you can write a series of simple notations to describe what’s important to recognize in your code in order to detect a dependency. A GrammarSketch production looks like this:

```
Java:SimpleVariableDeclaration:->
type reference ';' <-
```

Java indicates which language this pattern is for. GrammarSketch currently recognizes three languages:

- Java
- XML
- SQL

Unfortunately, the tool is case sensitive, so they need to spelled just like they are in the list.

¹Material presented in this section was originally published online at http://pages.cpsc.ucalgary.ca/cossette/GSK2008/gsk.html
SimpleVariableDeclaration refers to the production’s name. You can call your production whatever you want, it’s just a mnemonic device to help you later understand which of your patterns is finding a dependency in your code at a given point. It must be all one word.

The colons \( : \) and arrows \( \rightarrow, \leftarrow \) are just delimiters that are used to separate each section, but need to be present just like above: no spaces between anything on lines 1 and 3. Sorry, but it made it easier to write the parser. =)

type reference \( ':' \) is the pattern that you want to have matched in the source code. The rest of the instructions are organized into a series of examples explaining the basic productions available to you, and what they do.

D.2 Types vs. References

Generally what you care about in source code when looking for dependencies are identifiers - variable names, method names, class names etc. GrammarSketch uses two patterns to recognize identifiers as being either a type, or a reference.

A type is any identifier that has a meaning that’s the same regardless of context. For example, if you see

```java
String foo;
```
in one Java class, and later

```java
String bar;
```
in a different Java class, you know that String is referring to the same thing (the String class) in both cases even though they’re in different contexts.

A reference is the opposite - an identifier whose meaning changes depending on its context. For example, if you see

```java
String foo;
```
in one Java class, and later
in a different Java class, you know that even though the same identifier foo is being used, they’re not the same: in the first case, foo is referring to a String declaration, in the second, an ArrayList.

D.2.1 So Why is This Important?

GrammarSketch uses the concept of types and references to find dependencies. Instead of you having to describe what constitutes a dependency, GrammarSketch instead asks you to describe to it what is a type and a reference in each language. Based on that information, it tries to infer what the dependencies are. To see how to do that, look at the first example.

D.3 Example—The type and reference Patterns

In the Types vs. References section, we briefly explained what the difference is between these two things, and why GrammarSketch needs to know about them. Here, we show you how to write patterns to match types and references in your code.

Suppose we have the following Java code snippet:

```java
String bar = FooBar.someStringField; ArrayList foo;
```

String and ArrayList are both classes, or types, while bar and foo are both references. We could write a pattern to match this code as follows:

```
Java:SimpleIdentifier:->
  type reference
<-  
```

type and reference are special identifiers in GrammarSketch, and have two properties:
1. They try to match any C-style identifier in the code; any word which is comprised of letters of any case, numbers, and underscores.

2. They mark that identifier as important for detecting dependencies. GrammarSketch will store whatever that pattern matched and use it later for finding dependencies.

   In this case, the pattern is looking for two consecutive words, and will mark the first word it finds as a type, and the second as a reference. So String and ArrayList will be stored as types, and bar and foo as references.

   FooBar.someStringField will not be matched by this pattern because it is not two consecutive words - a dot (.) is between FooBar and someStringField. Later examples will show you how to deal with that.

D.4 Example—The Anchor Pattern

Suppose we want to capture the actual name of the following method declaration:

```java
public void fooBar( Bar b ) {
    String p = "Blah!";
    System.out.println( b.someMethod() + p );
}
```

If we were to use just a basic type pattern like:

```java
Java:SimpleIdentifier:--> type
<-```

We end up matching most of the text in the method. We will match fooBar, but we’ll also get a lot of other junk we don’t want (public, void, Bar, b, String, p, System, out, println, someMethod, p). We could instead do:

```java
Java:MethodIdentifier:->
type '('
<-
```

Any symbol or word between single quotes i.e. ’(’, ’public’, ’;’ etc. is treated as an anchor pattern; in other words, it must be explicitly matched in the code. So using this pattern, we’ll end up matching only those cases where an identifier is followed by an opening parenthesis, which are:

- fooBar
- println
- someMethod

If we really only cared about the method declaration, and were willing to be really restrictive, we could write

```java
Java:MethodIdentifier:->
'public' 'void' type '('
<-
```

and that would only match fooBar. Why Not Write ’public void’ Instead of ’public’ ’void’? GrammarSketch is flexible with whitespace - if multiple spaces, tabs, or even newlines are between public and void, it will ignore them if you write ’public’ ’void’. However, if you
write 'public void' it will expect to only find one space between public and void because that’s exactly what’s in your pattern. In this case, they’d both work, but one is more flexible.

D.5 Example—The STRING_type and STRING_reference Patterns

Especially with polylingual code, it’s common to have string literals in your code where what’s inside the string is important. By adding the prefix STRING_ to type and reference, you can capture the contents of those strings. For example, suppose we have:

```java
String classType = "ArrayList";
```

The pattern

```
Java:StringReference:-> type reference '=' STRING_reference <-
```

will match String as a type, classType as a reference, and ArrayList as a reference.

You could do something similar by writing the following pattern:

```
Java:StringReference:->
type reference '=' '"' reference '"'
<-    
```

The only difference is that this pattern will only grab a single identifier between two quotes. By using the STRING_ prefix, you capture the entire content of that string, regardless of how many words are in it.
D.6 Example—The [junk] Production

Generally what’s important for you to recognize in source code to detect a dependency is a small fraction of everything that’s there. GrammarSketch automatically ignores code that doesn’t match or fit your patterns, but can’t ignore stuff in the middle of your patterns. The [junk] production helps with this.

Suppose you have the following snippet of code:

public class Snafu extends AbstractFu implements IFoo {}

Suppose (for the sake of argument) we wanted to grab the class declaration, but also wanted to know what Interface it implemented. We need a way to discard everything after Snafu until we hit the implements clause. We could write:

Java:LongWayToGetClassAndThrows:->
'public' 'class' type 'extends' type 'implements' type '{'
<-  

This works, but it’s long and grabs some things we don’t want (AbstractFu). However, if we know that between any two points we can ignore whatever code is there, we can write:

Java:SimpleWayToGetClassAndThrows:->
'public' 'class' type [junk] 'implements' type '{'
<-  

The [junk] production ignores everything it sees until it hits the anchor token specified after it (in this case, implements).
Important Safety Tip: The [junk] needs to know exactly where it stops. Because of this, it must be followed by an anchor e.g. `implements`, `(', anything in single quotes. If it doesn’t see this e.g. if you were to write type [junk] reference, you’ll get an error message which may or may not make much sense to you (sorry, error handling for the notation is shaky at best).

D.7 Dependencies in OpenBravo

D.7.1 Quick Overview: Dependencies and GrammarSketch

A dependency is just a relationship between two pieces of code where if one piece of code changes, another will likely be affected in some way. GrammarSketch thinks about dependencies in terms of types and references:

- Types are identifiers that are universal in the code.
  
  For example, if you see a variable declaration with the type String in one Java file, and String in another Java file, it’s a safe bet that they’re both referring to the same thing - the String class.

- References are identifiers that only have meaning in a specific context.
  
  For example, you might see the variable String foo declared in one method, and the variable ArrayList foo declared in another method. foo is the same identifier in both methods, but it represents different object types in each case.

In general, the easiest way to think about them is:

- It’s a type if the meaning of the identifier is the same regardless of which file it’s found in.

- It’s a reference if the meaning of the identifier largely depends on its context.
D.7.2 Quick Overview: OpenBravo’s Languages and Structure

OpenBravo’s back-end uses three languages: Java, SQL, and XML. It uses a data-mapping technology to allow Java files to interact with SQL tables and queries by using XML to map the SQL statements to classes, methods, and fields that can be accessed in Java. The exercises in this study revolve around finding dependencies between Java, XML and SQL code.

This study was constructed with the understanding that you’re not an expert on OpenBravo. We picked OpenBravo because we’ve previously worked on it, and have some experience in identifying dependencies in the system. To help you out, we’re going to detail below what you’ll want to recognize in each language to detect dependencies in the system.

D.7.3 Quick Overview: Dependencies to Identify

For easy reference, here are key characteristics of each language you’ll want to recognize in order to find dependencies in the source code:

- **Java**
  
  - Type Declarations, e.g.
    
    * String foo;
    
    * ArrayList bar = new ArrayList();
  
  - Fields, e.g.
    
    * foobar.bar = "text";
  
  - Method Invocations, specifically static method invocations, e.g.
    
    * FooBar.invokeFoo( bar );

- **SQL**
  
  - Table and Column pairs, e.g.
• SELECT foo.bar

• XML

  – Attribute / Value pairs

    There’s one specific attribute that’s worth recognizing:

    name = "someMethodName"

    The value associated with the attribute name is often used elsewhere in the Java code to access an associated SQL query.

  – Element tag names

    There’s a couple of XML element names that are significant:

    * <SqlClass/>
    * <SqlMethod/>

    The element names themselves don’t matter, but if you’ll find a name attribute later in that tag whose value maps to a Java Class or Method respectively. You can use this to limit which name attribute/value pairs you recognize.
Appendix E

GrammarSketch Configurations by Participants

E.1 Participant 1

Java:StringTest:->
  type reference '=' STRING_reference
<-

Java:MethodIdentifierTest:->
  reference '.' reference '('
<-

Java:SimpleTypeTest:->
  type 'ArrayList'
<-

SQL:PairsTest:->
  'SELECT' reference '.' reference
<-

XML:NameAttributeTest:->
  'name' type
<-

132
E.2 Participant 2

Java:stringFoo:->
  type
  <-

Java:fooBar:->
  "." reference
  <-

Java:methodInvoke:->
  type "." reference
  <-

SQL:tableColumnPair:->
  type "." reference
  <-
E.3 Participant 3

Java:SimpleIdentifier:->
type reference
<-

Java:ObjectCreationIdentifier:->
type reference ' = ' 'new' [junk]' ('
<-

Java:FieldInstance:->
type '.' reference ' = ' STRING_reference
<-

Java:MethodInvokation:->
type '.' [junk]' ('
<-

XML:attributeValuePair:->
' name ' '=' STRING_reference
<-

XML:tagReference:->
' '< reference '>'
<-
E.4 Participant 4
Java:TypeDeclarationConstructorInitialized:->
type reference '=" new type [junk] )" ;'
<- 

Java:TypeDeclarationStringInitialized:->
type reference '=" [junk] "" "" ;'
<- 

Java:TypeDeclarationEmptyStringInitialized:->
type reference '=" " " " ;'
<- 

Java:Field:->
reference "." reference
<- 

Java:FieldAlternative:->
type "." reference
<- 

Java:MethodInvocation:->
reference "( [junk] )"
<- 

Java:MethodInvocationFromClassOrObject:->
E.5 Expert Configuration

SQL:TableColumnPattern: -> type "." reference
<-
type [junk] 'join'

XML:ClassTypeDeclaration:->
'<' 'SqlClass' 'name' '=' STRING_type

XML:MethodInvocationMapping:->
'<' 'SqlMethod' 'name' '=' STRING_type

Java:TypeDeclaration: ->
'class' type [junk] '{'

Java:StaticMethodInvocation: ->
type '.' type '('

Java:MethodInvocation: ->
reference '.' type '('

Java:ArrayDeclaration: ->
type '[]' reference
Java:ObjectArrayInvocation: ->
reference '([' [junk] '])' '.' type
<-