DSketch: Lightweight, Adaptable Dependency Analysis

Brad Cossette
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
cossette@cpsc.ucalgary.ca

Robert J. Walker
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
walker@ucalgary.ca

ABSTRACT

Software developers who extend or repair existing software systems spend considerable effort in understanding how their modifications will require follow-on changes in order to work correctly. Tool support for this process is available for single, popular languages, but does not suffice for less popular languages, uncommon language variants, or arbitrary combinations of languages and connection technologies. We have created the DSketch tool so that developers can create a lightweight pattern specification for how dependencies can be heuristically identified in their systems. We performed two case studies involving industrial developers who applied our tool for conducting polylingual dependency analysis in software systems; the developers found it easy to configure the tool for their needs, were able to adapt their patterns to new contexts, and had sufficiently accurate dependency predictions for their work.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering.*

General Terms

Design, Human Factors, Languages, Management.

Keywords

Dependency analysis, polylingual systems, pattern matching, approximation, developer feedback, lightweight tool support, DSketch, case study.

1. INTRODUCTION

To avoid introducing errors into their software systems, developers conduct *change impact analysis* to understand the consequences of any modifications they enact [1]. Change impact analysis must deal with the *dependencies* [19] in the system; the presence of a dependency indicates the possibility that a change to the depended-upon entity may require a change to the dependent entity—this is the dreaded "ripple effect" [20]. Unfortunately, developers are surprisingly poor at identifying dependencies manually [9], and tool support is often unavailable and too expensive to create.

Tool support for dependency analysis is ideal when it is aware of the detailed semantics of the programming language used in a particular software system: the presence of dependencies follows on directly. But such tools are tied to a particular programming language—and worse, a particular version thereof—making them hard to adapt to new languages or contexts without re-implementation [17]. In the context of software systems where multiple programming languages are in use [10] (which we refer to as *polylingual software systems*) the problem is further complicated: in addition to coping with each language in use, each combination of programming languages can use varying semantics to describe interactions across language boundaries [6], leading to a "combinatorial explosion" of situations that may need support [3]. The identification of the dependencies either becomes an onerous task of interprocedural analysis or depends heavily on the specific protocols used by the connection technology (e.g., JNI for communication between Java and C++ source code, embedded SQL queries, reflective configuration via XML files). Sound analysis becomes impracticable or even undecidable in such situations [16, 12].

To overcome these difficulties, we propose a lightweight approximation of semantically-aware tool support (1) to provide sufficiently accurate dependency detection, (2) to provide tool support for dependency detection that can be adapted to arbitrary language and technology combinations, and (3) to require significantly less effort on the part of the developer who needs to construct the tool support. Our approach is embodied in the DSketch tool, which leverages the developer's knowledge about the syntax and semantics of the technologies used in his software system.

To use DSketch, the developer writes simple pattern specifications that should select identifiers involved in dependencies. A set of heuristics is applied by the tool to find correspondences between these identifiers, which represent probable dependencies in the system. The result is displayed to the developer by annotating the source code[1] to indicate which pattern matched which identifiers; DSketch is built into the Eclipse integrated development environment, allowing the developer to continue to use their other tool support

[1]We use the term "source code" liberally here to include XML files, configuration text files, etc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

as desired. The developer can iterate on the pattern specifications to refine the set of dependencies until satisfied with the results. The workflow intended for the tool involves one developer configuring a set of patterns for a particular system, and allowing other developers to use these patterns in their tasks on the same (or similar) systems.

We conducted two case studies with industrial developers to investigate the strengths and weaknesses of the approach, particularly with respect to the intended workflow. We had the first developer use DSketch to identify the dependencies on a subset of an industrial system; we then had this same developer apply and adjust the patterns on other parts of the same system. The second developer took the patterns of the first and adjusted them as needed to recognize dependencies in another system which used the same programming languages, but differing technologies, and used the tool to assist their investigation of a change task.

The remainder of the paper is structured as follows. Section 2 provides a small motivational example to illustrate the problem of polylingual dependency analysis. Section 3 describes the related work, and points out that our problem has not been solved previously. Section 4 presents details of our proposed solution. Section 5 describes the case studies exploring the usefulness and usability of our proposed solution. Section 6 discusses remaining issues and future work.

This paper contributes a novel approach to lightweight tool support for polylingual dependency analysis.

2. MOTIVATION

Consider a developer who is to create an online web-store for her company's employees to access over an internal network. The intent is to allow employees to browse and order company-branded clothing and merchandise, either as a reward for service or at a subsidized rate to promote company loyalty and identity. Because this is a small, internal application, the developer believes that it might be worthwhile to adapt an existing system to her needs, specifically the iBatis JPetStore v5.0 reference application (a small-scale web-store system that ships as an example with the iBatis data mapping framework). She knows that the login mechanism used by JPetStore will need to be replaced with one that leverages the company's existing network login credentials for each employee. Thus, she must estimate the impact of this change on the rest of JPetStore.

JPetStore uses a series of XML files to define an object–relationship mapping between Java classes and SQL queries, for which the iBatis framework can then generate Java objects at run-time. Using a text editor, the developer looks for the `sql-map-config.xml` file, which defines which XML mappings are in use, and decides that `Account.xml` may be the most relevant. Browsing the queries in the file, she 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 1.

As a first attempt, the developer runs a case-insensitive lexical search on the entire system to find other references to the `USERNAME` column; the search returns 106 matches. Looking through the results, she believes that these are mostly false positives so she fails to examine them closely (as a result, she overlooks references from within the `Order.xml` file). She restricts the search to just the `Account.xml` file: three other queries that access the `USERNAME` column are

```
<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 1: JPetStore's XML specification to add a new user's credentials.

found. The developer is reasonably confident that these are the only three queries that she has to worry about, as the names of the other XML files do not suggest that they deal with user authentication. Her next step is to identify what Java functionality is dependent on these queries. To do this, the developer must understand the semantics of the iBatis framework in mapping a query to a Java method. As an example, in Figure 1 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. The developer determines that a total of four methods in the `AccountSqlMapDao` class are dependent on the authentication queries (two `getAccount(..)` methods and `insertAccount(..)` are the others).

To continue her 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 IDE like Eclipse, she can use the semantic tools it provides to quickly infer what classes are dependent on these methods. She determines that three classes are dependent on the four `AccountSqlMapDao` methods: `Account`, `AccountBean`, and `AccountService`.

The developer now has a problem: she knows that several web-pages (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 those are, or on which entities in the Java code the functionality depends. The developer resignedly sets up a Tomcat server on her machine, deploys the JPetStore system, and manually investigates the web-pages. She guesses that `NewAccountForm.jsp`, `EditAccountForm.jsp`, and `SignOn.jsp` are the most relevant. These web-pages are written using Java Server Pages (JSP), and make use of yet another framework called Apache Struts to dynamically generate content.

Getting annoyed at the time this "estimate" is taking, the developer opens the three webpages in her IDE. Again, an understanding of how the Struts framework encodes dependency information is needed for the developer to manually and precisely determine what dependencies exist between each JSP page and the Java source code. In Figure 2, 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, the developer may correctly infer that `"accountBean"` is referring to the `AccountBean` class, which she identified earlier as depending on the authentication queries. But a lexical search on the fields within

```
<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: Excerpt from `NewAccountForm.jsp`.

`AccountBean` will not yield a match for `username` or `password`. The developer must remember that the Struts framework supplies "TagLibs" which allow an implicit mapping of the property attribute here to get/set *method* invocations on the `AccountBean` class (e.g. `getUsername`, `setUsername`), and alter her search accordingly. 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.

What the software developer wanted was to select a few queries that she thought would change, and to 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, she 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 obtaining her 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.

3. RELATED WORK

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 differing definitions of dependencies could have [19, 16, 12]. More pragmatic approaches to impact analysis followed [1], but still made strong assumptions about having the intended change implemented in order to perform the analysis.

The problem of conducting dependency analysis in a polylingual context is not new; 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 [5]). Some protocols are independent of the implementing languages, such as web services and the Common Object Request Broker Architecture. Others target specific language combinations, like the Java Native Interface which bridges Java and C/C++ code [13, 7]. The semantics describing the cross-language dependencies in each system are often specific to the technology used, and not generalizable.

Lexically-based dependency analysis tools permit the developer to define patterns, often based on regular expressions, to recognize dependency structures in code; the most well-known is the grep [4] family of tools. Regular expressions are flexible and descriptive enough to be useful for polylingual systems, yet they possess significant usability issues in practice that result in their poor comprehensibility and modifiability. As an example, consider the regular expression in Figure 3, used to describe the start of a method declaration (not including modifiers) in Java. There are three errors in this expression that we now know of, despite multiple sets of eyes having looked at it over an extended period. These errors are likely not obvious to a developer under pressure to complete a task quickly (or to you).

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

Figure 3: The arcane nature of regular expressions.

A more refined approach is the lexical source model extraction (LSME) tool [17], which improved upon several of grep's basic limitations including supporting hierarchically structured patterns. However, LSME's notation language is similar in design to regular expressions, and requires more information than perhaps is necessary if it were strictly focused on dependency analysis. Further, it also requires that the user specify how pattern matches must be operated on since it is not limited to dependency analysis. Atkinson and Griswold [2] created the TAWK tool that requires the user to supply an abstract syntax tree (AST) specification for a language and AWK patterns to match AST nodes to source code elements. This approach is aimed at generating a complete AST for a source code system, and requires more information than may be minimally necessary to estimate structural dependencies.

Syntactically-based tools for polylingual systems (e.g., [13, 7]) are typically focused on the technologies that bridge languages for interoperability, and are configured for specific language combinations. 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. Some tools can further leverage their understanding of a language's semantics to support natural language querying against semantically-meaningful identifiers in source code [8]. 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, which is a non-trivial task [14]. 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.

Techniques for fault-tolerant parsing have been promoted, such as fuzzy parsing [11] and the island grammars approach [14]. The latter allows certain details not of interest (such as the modifiers in our regular expression above) to be ignored. We have previously applied the island grammar approach in order to detect dependencies in polylingual software systems [6]. We created a testbed called Luther to analyze the cost-to-accuracy relationship between a series of island grammars. Our results for that study suggested that

the effort involved in writing more accurate island grammars rises faster than the resulting accuracy. We have not previously applied those results to define a specific approach for lightweight dependency analysis.

4. APPROACH

Our approach, embodied in the DSketch tool, relies on the developer to "sketch out" the patterns needed to recognize relevant dependency syntax in the source code; in return, DSketch uses these patterns to extract identifiers, to delineate their contexts, and to apply a heuristic analysis to tell the developer where dependencies exist in their system. Our previous work with the Luther testbed [6] identified several issues that made it difficult to adequately provide lightweight tool support that could be configured by developers for polylingual dependency analysis, and our tool design incorporates this feedback by seeking to minimize the developer's effort as much as possible, sometimes sacrificing precision as a tradeoff. The rest of this section is organized as follows: in Section 4.1, we examine the language we provide to the developer for sketching patterns. We then explain how DSketch uses these patterns to detect identifiers in the source code in Section 4.2. Section 4.3 describes our heuristic approach for interrelating identifiers to suggest the presence of dependencies. Finally, we describe the developer's experience in applying DSketch, in Section 4.4.

4.1 Sketch Language

Our sketch language borrows several concepts and ideas from work on fuzzy parsing [11], island grammars [14, 15], and our previous work with Luther [6]. There are four key design principles which shaped the development of our notation: (1) human readability is paramount; (2) details that hinder the expression of intent should be eliminated; (3) allow implicit and explicit ignoring of irrelevant code; and (4) use "anchor points" to help deal with ambiguity [11].

The sketch language comprises four main components:

Identifier keywords. DSketch is primarily concerned with extracting identifiers from source code on which dependencies may be based. Two keywords are provided for this purpose: `global` and `local`. Each of these keywords matches a sequence of characters that lexically conforms to a C-style identifier. The different keywords are provided to allow the developer to give a very crude approximation for scope resolution, by indicating if dependencies involving this identifier should be sought across the developer's entire source code base (`global`), or only within the file (`local`).

Anchors. Patterns can require specific lexical sequences to be embedded within them to improve their ability to discriminate; these are called *anchors*. Any string or character that serves as an anchor is enclosed in single quotes (e.g., `'public'`, or `'{'`). A backslash is used as an escape-character (i.e., `'\\'` matches a single backslash character and `'\''` matches a single quotation mark). Anchors allow developers to provide additional pattern information to improve match precisions, without confusing the tool as to what portions of the match need to be analyzed for dependency analysis.

String literals. It is quite common for source code and important identifiers from one programming language to be embedded in the source code for another language, typically as a string literal. In addition to anchor patterns described earlier, the developer can add the `STRING_` prefix to the identifier keywords (e.g. `STRING_global`) to instruct DSketch to grab the entire contents of a string literal and treat it as having the scope of the suffixed identifier.

Junk productions. Some patterns can be difficult to specify in detail, because of the various syntax combinations that may exist in the source code; this is particularly onerous when these details are extraneous to the purpose of the developer. Rather than anticipate all such combinations, the developer can use a *junk production* to instruct DSketch to ignore everything it finds at that point in the pattern. The syntax of a junk production is `[junk]`; it must be followed by an anchor to give DSketch a concrete means of determining when the irrelevant material ends. The junk production accepts every character it encounters until it reaches the first match for the anchor.

4.1.1 Example

As an example usage of the sketch language, consider this code snippet from the `MenuData.xsql` file in OpenBravo:

```
<SqlClass name="MenuData"
package="org.openbravo.erpCommon.utility">
```

To match the identifier in the `package` attribute, the developer can sketch this pattern:

```
'<' 'SqlClass' [junk] 'package' '=' STRING_global
```

The anchors `'<'`, `'SqlClass'`, `'package'`, and `'='` are all used to describe the characteristics of an XML tag element, but also describe a specific tag element that has special meaning (in OpenBravo, the `SqlClass` tag defines the Java class to which the encapsulated SQL queries will be mapped at runtime). Here the developer is not interested in capturing the `name` (or any other) attribute, so he uses the `[junk]` production to skip over anything intervening, and then uses the `STRING_global` keyword to capture the contents of the string literal and mark it as having global scope.

4.2 Detecting Identifiers

DSketch compiles the developer's sketched patterns into a set of equivalent regular expressions. The tool traverses the source code, determines which patterns need to be applied to each source file based on its extension, and attempts to match the patterns. Matches for the identifier keywords and details of their position are stored within a plugin-hosted database. Once this process is completed for the entire code base, the database 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 which patterns discovered them.

4.2.1 Compiling Sketched Patterns

DSketch uses a parser we created using JavaCC to analyze the patterns sketched by the developer, and to transform them into a set of regular expressions. The transformation rules are as follows:

- Occurrences of `global` and `local` are expanded into a regular expression appropriate for matching C-style

identifiers. This regular expression is enclosed in parentheses to allow back-referencing to be used to recover the identifiers matched in the pattern.

- Occurrences of the `STRING_`-variant keywords are used to match the entire contents of a string literal, and then record its matches as a global or local identifier.
- Anchors are embedded into the regular expression, with appropriate escape-characters inserted when necessary to avoid their interpretation as meta-characters.
- A junk production (which must be succeeded by an anchor) is translated into a regular expression that permits arbitrary characters but that will greedily match the anchor.
- The presence of whitespace is dynamically calculated based on the combination of productions in the DSketch patterns. The rules are as follows:
  - No whitespace characters match at the start or end of the pattern.
  - One or more whitespace characters must match between any two consecutive identifier keywords.
  - No whitespace is matched between a `[junk]` production and its succeeding anchor.
  - Between an anchor and any other element, zero or more whitespace characters may be present.[2]
  - Between any two other elements, zero or more whitespace characters match.

DSketch then uses the regular expression engine provided with the Java 6.0 Standard Edition SDK to apply these regular expressions to the developer's source code. For each file, all regular expressions appropriate to the programming language(s) contained therein are applied, and matches are extracted. For each match, DSketch stores: the match's location; the name of the pattern that found the match; the identifiers in the match tagged as global or local; and in the case where multiple identifiers were located from the same match, annotates these identifiers as *siblings*.

4.2.2 Pattern Collisions

It is possible to sketch two or more patterns that match the same portions of a file; perhaps the sketch is wrong, or it may reflect the difficulty of writing patterns for that programming language's syntax. Such a *pattern collision* may cause the same identifier in the source code to be recognized by multiple patterns. We cope with pattern collisions in two ways. (1) In cases where one pattern marks an identifier as local, and another marks it as global, we prefer the pattern which recognized the identifier as global to reduce the risk of false negatives in the analysis. (2) If multiple patterns all recognize the same identifier with the same scope (e.g., all global), we associate the identifier with the first pattern match, and ignore subsequent matches.[3]

[2] This can cause a false match in some cases (e.g., `'public' 'void'` matches `publicvoid`), but we feel that this situation would occur rarely in practice.

[3] In earlier iterations, we merged the meta-data associated with each pattern collision, and showed the developer all the patterns which matched a particular identifier. This seemed to confuse them more than it helped.

4.3 Detecting Dependencies

Once a set of identifiers have been identified, DSketch applies a set of simple heuristics to approximate how these identifiers indicate dependencies in the system. The assumptions that DSketch makes about how dependencies are formed reflect both our attempts to provide a simplified notation scheme for the developer to use, and our own experiences in dealing with polylingual software systems. The following heuristics are currently used by DSketch:

- Dependencies can only exist between identifiers extracted from source code through one of the developer's patterns. That is, DSketch does not search the source code to see if matches can be found outside of the pattern set the developer provides.
- Dependencies must be lexically identical, ignoring case.
- If the identifier was classified as `global`, it is dependent on all other occurrences of that identifier, regardless if they were marked as `global` or `local`.
- If the identifier is classified as `local`, it is dependent on all other occurrences of that identifier that (a) are marked as being `local` and that (b) are discovered within the same source file. The identifier is still also classified as being dependent on any lexically identical identifiers which are classified as `global` that occur elsewhere in the system.

These heuristics derive from our experience that many polylingual systems rely on lexical identity in some form as the primary means of indicating where a relationship exists.

4.4 Using DSketch

Figure 4 shows a screenshot of the DSketch dependency analysis tool, installed as a plugin to the Eclipse IDE. DSketch is intended for use by developers looking to investigate dependency relationships, particularly in a polylingual system, where a reasonable approximation of the dependencies is sufficient for their needs. A developer using the plugin will have a project in Eclipse with the complete set of source code for the software she intends to analyze.

The developer begins with a view (perhaps empty) of the sketches currently defined for her system, as shown by the Production Table View (see marker A in Figure 4). She can add to the set of patterns (or edit existing patterns) by editing or creating new sketches in the Sketch Editor window (see marker D). If she desires, she can disable individual patterns so that DSketch does not use them to either extract pattern matches, or conduct analysis; we have found this capability valuable in debugging sketched patterns. She can also adjust the languages associated with various file types in her source code base through the language mapping editor (see marker B).

If she is interested in seeing where points of dependency may exist in a file, she uses a right-click context menu in her source code editor to highlight all of the pattern matches in the current source code file (see marker C); DSketch then highlights identifiers in the file, colour-coding them to indicate which pattern matches at that location, as well as setting a tooltip on the identifier with the same information. For example, in Figure 4, the "XMLMethod" sketch

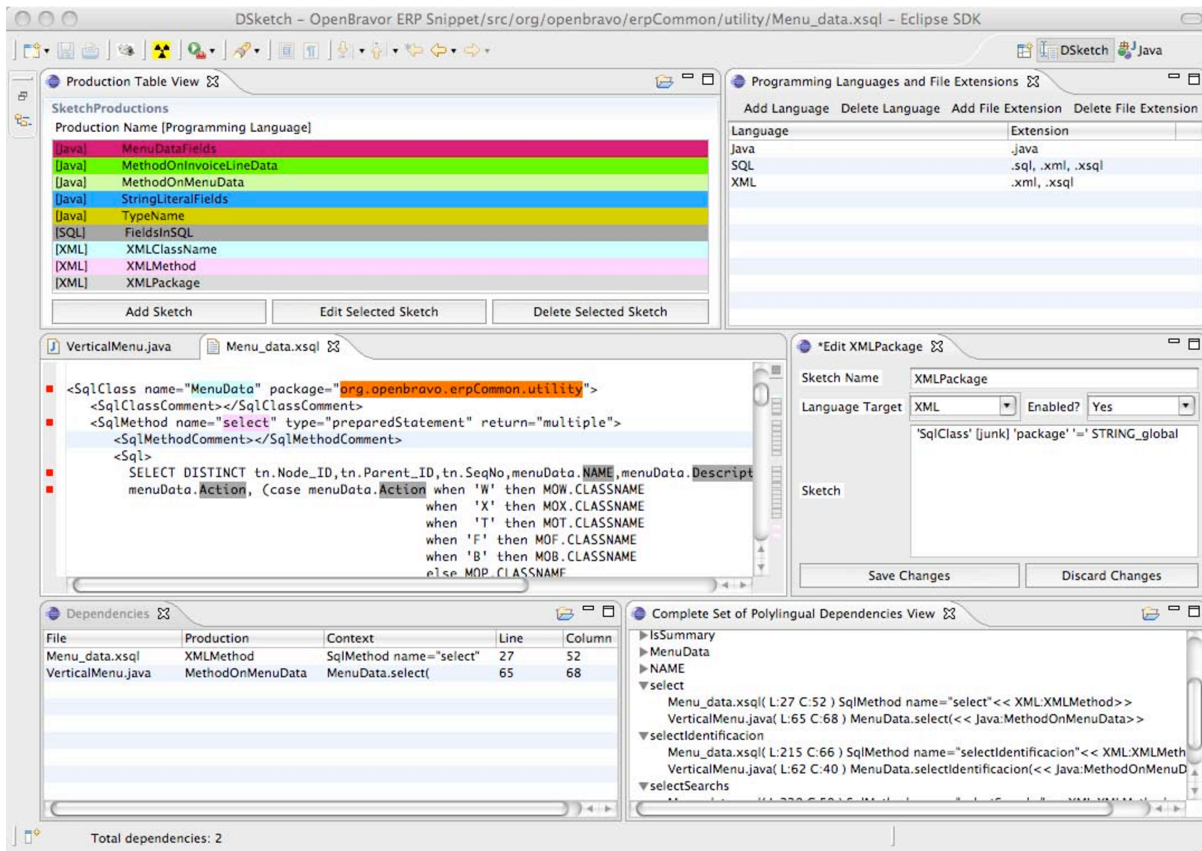

Figure 4: Screenshot of DSketch.

has extracted an identifier from the **name** attribute in the **SqlMethod** tag, and has marked the identifier with the colour pink (see marker C) to indicate (1) that it has been matched, and (2) that the "XMLMethod" sketch has identified it as relevant. If the developer right-clicks on the "select" identifier, they can use a context menu to have DSketch display polylingual dependencies on this identifier (see marker E), which here shows a match to a static method invocation in Java. She can also view the entire set of dependency predictions for her system if she wishes to have a sense of the entire scope (see marker F). In either view, each entry shows the identifier matched, its location, and the context surrounding the match. She can leverage this information to initially discriminate between the dependencies, and to find those dependencies that are more relevant for her investigation task. If the developer sees a dependency that she wishes to explore, she can double-click on it to automatically navigate to the appropriate location in a source code editor.

5. EVALUATION

The intent of DSketch is to provide developers with adaptable polylingual dependency analysis tool support that is relatively easy to configure for a system, and sufficiently accurate to justify its use over state-of-the-practice approaches. We conducted a qualitative investigation of the tool to determine whether it meets these goals. Our specific research questions were as follows: (RQ1) "Can developers successfully configure the tool for dependency analysis?"; (RQ2) "How much effort is required to adapt the tool to a new system?"; (RQ3) "Is the tool sufficiently accurate to be useful for developers?"; and (RQ4) "How does our approach conform with or contradict developers' expectations?"

To address these questions, we conducted two case studies in which industrial developers used DSketch to perform dependency analysis in two separate software systems.

5.1 Case Study Overview

Two industrial software developers were recruited to participate in the study. Each of the participants was asked to investigate the polylingual dependency relationships between two source code files in a particular system. Case Study 1 focused on the open-source OpenBravo ERP system, while Case Study 2 involved the iBatis JPetStore v5.0 reference implementation. Both of these software systems use a data-mapping framework to allow Java source code to interact with SQL queries and results, by embedding SQL queries within XML tags that describe how Java code should interact with them. However, each uses a different technology with similar yet distinct semantics for facilitating these operations: OpenBravo uses (in part) a compiler called SQLC, while JPetStore uses the iBatis Data Mapper framework.

Each case study was structured as follows: After a short tutorial explaining the mechanisms that the particular system uses to map dependencies across these languages, the

participants were asked to manually investigate a portion of the source code of the system and determine what were the polylingual dependencies present. Each was provided with the Eclipse IDE, v3.5. The goal was to have the participant become sufficiently familiar with the system that they would be able to later judge the efficacy of their DSketch configuration in detecting these same dependencies. Since the developers were unfamiliar with the details of the particular systems under analysis, they were provided with training material and permitted to ask the researchers questions concerning dependency mechanics over the course of the study, as a means of simulating the experience each would have when working with their own systems.

After the developer declared himself as being done with the manual investigation task, he was presented with our tool. After a training session in which the tool's operation, sketch language, and behaviour were explained, each participant was given time to experiment with the tool on a piece of sample code to become familiar with writing patterns. Once he felt comfortable with the tool, he was given his assigned system to investigate. After their investigation was complete, the participant was interviewed about his experiences with DSketch.

5.2 Case Study 1

The participant here described himself as having 16 years of industrial experience as a software developer, familiar with maintenance of polylingual software systems, and for the last several years as working as a principal developer and architect on a polylingual software development kit used extensively in his company's and clients' products.

To constrain the task size, we asked the participant to manually investigate all the polylingual dependencies between just two files in OpenBravo: `VerticalMenu.java` and `MenuData.xsql` from the `org.openbravo.erpCommon.utility` package. These two files comprise 429 non-commented lines of code and extensively reference each other. After configuring the tool to his satisfaction, he was supplied with two additional files from the system (`InvoiceLine.java` and `InvoiceLine_data.xsql` in the `org.openbravo.erpCommon.info` package), and asked to adjust his configuration to recognize dependencies between them.

5.2.1 Manual Treatment

After being given demonstrations about the dependencies on some example files, the participant commented that he was accustomed to SQL/XML data mapping techniques, but each is slightly different, and so he was primarily concerned with understanding how the semantics of this particular technology worked. The participant started his investigation by examining the relationship between the XML tags and the Java code; he found tracing the various SQL queries and enclosing XML tags painstaking in the default text editor, and opened the `MenuData.xsql` file in Eclipse's XML editor to browse the contents as a tree structure.[4] With this view in hand, he easily determined the mapping of the `MENUDATA` table and its associated queries to the appropriate type static method invocations in Java.

The participant continued with the laborious task of identifying where SQL column references in the `MENUDATA` table were being accessed as fields on `MenuData`-typed objects, remarking at one point: "Is there an easier way to find all of these field names?" He noticed that a particular coding convention caused `MenuData` declarations to be labeled as "`menuData`", and he proceeded to perform lexical searches on this identifier to find the related field accesses. In doing so, the participant missed some occurrences elsewhere that failed to follow this naming convention (he did not discover this issue at the time). He then declared himself finished, having spent roughly 40 minutes in the investigation.

5.2.2 DSketch Treatment

The researchers then commenced the training of the participant in the use of DSketch, including practice at sketching patterns and interpreting the results. During training, two issues arose. (1) The participant asked if regular expression closure operators (e.g., +, *) are supported; they currently are not. (2) The notion of global versus local identifiers caused some confusion. When sketching a pattern to recognize the value associated with an XML attribute, he debated as to which scope made more sense: the XML language defines that the value of an attribute only has meaning within the context of its enclosing XML tag element, which would imply that the value should be marked as having a local meaning. However, OpenBravo uses the value associated with the name attribute in certain tags to define which static method invocation the enclosed SQL query should be mapped to, indicating that this attribute value has a meaning outside of the file, and should be marked as global; the participant eventually recognized this.

The case study was suspended at this point at the participant's request, and was resumed the following day. The participant spent approximately 15 minutes reviewing the test patterns sketched the previous day. He then chose to delete these patterns before starting the configuration task.

The participant started by sketching three patterns quickly (one for each of the languages in use, to detect a particular cross-language dependency), and testing them against the `VerticalMenu` class. In general, this participant tended to pick a particular snippet of code that was typical of the dependency characteristic he wanted to recognize, and to sketch a pattern designed to specifically recognize that snippet. Upon reviewing the results, he would then refine that pattern if it was too specific, or not specific enough, to match other locations in the source code until he was satisfied with the pattern's coverage. Primarily, the participant wrote patterns to recognize table and column references in SQL, attribute values associated with XML tags which encapsulated SQL queries, static method invocations on the `MenuData` class in Java, and field accesses in Java.

The participant was then given the second set of files from the OpenBravo system, and asked to see what adjustments needed to be made to recognize dependencies in this files. The patterns written for the SQL and XML code did not need to change, but his set of Java patterns needed to change in two ways: (1) The participant originally sketched a pattern to recognize static method invocations specifically on the `MethodData` class, as these represented SQL queries; in the new files, similar support had to be added for the `InvoiceLineData` type. (2) The `InvoiceLine` class has numerous string literals that contain references to SQL tables and columns, a feature not present in the previous file set. After making these modifications, he was satisfied with the results. He had spent approximately 30 minutes configuring

[4]The file extension had to be changed to work around the editor's constraints.

DSketch for the first set of files, and an additional 10 minutes to adapt the patterns to the second set of files.

5.2.3 Observations on Workflow, Success, and Effort

The participant tended to iteratively supplement his configuration by sketching a new pattern, testing it, and adjusting it as needed before moving to the next. He worked through one language at a time, capturing all the relevant syntax before moving on. He also tended to sketch his patterns as narrowly as possible; he said that his attitude was "Garbage-in, garbage out. A tool is only as good as its inputs."

Over time, the participant seemed to gain greater trust with the tool as he investigated predicted matches. In the first set of files, he noticed that DSketch did not find a match for one of the SQL queries. After investigation, he realized that this omission was correct—that query was never accessed in the Java code. Later, a similar issue occurred with the second set of files, but he now accepted the tool's predictions without feeling the need to investigate.

All the sketch language features were used at some point by the participant, and with one exception were found useful. During the study, the `STRING_`-variant keywords were found to have an unintended consequence: these keywords match the contents of a string literal but implicitly assumes that the string is *not* empty. The presence of several empty strings in the second set of files caused the resulting pattern to match content between the last double quotation mark (`"`) in an empty string, and the first occurrence of a double quotation mark later in the file. The participant easily fixed this by eliminating the `STRING_`-variant of the keyword with its normal variant and appropriate anchor productions.

5.3 Case Study 2

The second participant described himself as having 7 years of industrial experience as a software developer, the vast majority of that involving polylingual systems. We asked the participant to manually investigate the change impact of altering the user authentication mechanism in JPetStore. To limit the required effort to a reasonable level for a case study, we restricted his investigation to just the Java, XML, and SQL portions of the system. After the manual investigation, he was trained on DSketch, and provided with the configuration generated by the Case Study 1 participant as a starting-point configuration for JPetStore.

5.3.1 Manual Treatment

During the explanation of how iBatis maps data objects in Java, the participant commented that he was familiar with this paradigm as the semantics seemed very similar to those used by the Hibernate data mapping framework, which he had used before. The participant constructed a change plan on paper to record his understanding of how the system would be affected by the anticipated change. He largely investigated the system by opening source code files, and scanning for relevant portions of code, pausing in some cases to thoroughly understand a particular segment that seemed to be of special significance to the authentication mechanisms. After some initial exploratory searching in this fashion, he returned to the `Account.xml` file which contained most of the SQL queries involved in the user authentication mechanism in JPetStore, and looked for dependencies emanating from the `SIGNON` table throughout the code. The participant would either use lexical searches, or Eclipse's Java tooling, to follow dependency links in the code, but occasionally would investigate a new file when its name or package identifier suggested it was of relevance. He continued this pattern of investigation until he reached the `AccountBean` class which interacts with the system's presentation layer, after which he finalized his change plan and declared himself finished. He had spent roughly an hour creating this change plan.

5.3.2 DSketch Treatment

The researchers then commenced training the participant on using DSketch, with the final configuration from Case Study 1 provided as a starting point. During training, he asked if the zero-or-one regular expression operator (i.e., ?) was supported; it currently is not. Once the participant felt comfortable with the tool, he began his configuration task for JPetStore, again with the patterns from Case Study 1 as a starting point.

The participant began by systematically examining the patterns, and had no problems reasoning about the intent behind the Java and SQL patterns. This changed when he came across the first of the XML patterns: after examining the pattern, he was confused about the intent as it described syntax that he did not recognize in their system. He manually examined the JPetStore code to look for relevant examples, and only after a few minutes of this did he remember that these patterns had been created to match dependency syntax which differed from that used by iBatis in JPetStore. He decided that the XML patterns were not useful for his context, but the existing Java and SQL patterns could be reused (even though at least one pattern described syntax which did not occur in JPetStore); surprisingly, he decided to not delete the irrelevant patterns since they seem to have no ill effect.

He continued his configuration of DSketch by applying the patterns to see what dependency syntax was missed. In the case of the SQL code, he noticed that the existing patterns captured most of the syntax he was interested in, but missed a few small cases where queries did not use explicit access of a table column (e.g., `SIGNON.USERNAME`) that the other patterns had leveraged. He decided in this case that the tool's configuration was sufficient since it already found relevant matches on those tables and columns in the same file, and felt it was unnecessary to write specialized patterns to capture these few cases. He also decided that only one additional Java pattern was needed so that field declarations were recognized, so that DSketch could match them against the SQL column names to which they were mapped.

In exploring the resulting matches, he discovered two false positives which he had not expected: DSketch indicated that there were dependency matches on the `USERNAME` and `PASSWORD` columns in the `sql-map-config.xml` file, which otherwise is expected to simply list which XML files contain SQL query mapping data for iBatis. DSketch's dependency preview indicated the context of these matches as being `JDBC.Username` and `JDBC.Password`, which was ambiguous: it could describe a Java field access, or a SQL table/column pair. After investigating the match location, the participant saw that these were actually describing property values which were to be supplied by the iBatis system for connecting to the JPetStore database.

5.3.3 Observations on Workflow, Success, and Effort

The participant's workflow was substantially different than

that of the Case Study 1 participant, because he intended to formulate a specific action plan for enacting a change to the system. Consequently, he spent a considerable amount of time attempting to understand how existing code functioned, rather than simply tracing dependency connections. Having an explicit change plan may also have explained why he did not feel it necessary to configure DSketch to recognize all the known polylingual dependencies in the system: those few dependencies were easier to simply make note of in the change plan, rather than invest additional effort to address. The participant felt comfortable reusing the patterns written by the earlier participant, and only needed to make minor alterations for his system's context.

In using DSketch's dependency preview window, the participant noted that the context provided for matches was insufficient: "Having the filename and the context helps, but I need to see the entire context." The context currently provided by the tool is the entire match (including anchor and junk productions) for that pattern, which in some cases he encountered was too limited to provide useful feedback. Had the preview included most of the code in that line, it would have been sufficient information to reject the dependency without needing to investigate the code snippet.

5.4 Analysis of Observations

Here we consider how our observations address research questions RQ1–RQ4.

5.4.1 Can developers configure the tool?

Both participants were able to configure DSketch tool to recognize polylingual dependencies in systems, despite a limited amount of exposure to the tool prior to use, and in spite of not being experts with the systems they were analyzing. In the case of the first study participant, he was able to return to his task a day later, revisit his previous work, and continue his configuration of DSketch within a 15-minute period. The second participant was able to take a set of patterns written by a different developer for a different system, understand the intent behind the patterns, and supplement them appropriately for his own work.

5.4.2 How much effort is required to adapt?

Both participants described the effort as non-trivial, but reasonable. The second participant was able to easily adapt the DSketch configuration for OpenBravo to the JPetStore system, by reasoning about what features the existing patterns were intended to match. He discovered which patterns were inappropriate, and appropriately supplemented the configuration with additional patterns to address what was lacking. The developer did note though that it was difficult to get an initial sense of what was missing in the pattern configurations; if the pattern set were sufficiently large, it could be difficult to understand the tool's effective coverage. The first participant commented that: "[DSketch] has an up-front cost that has to be paid in learning the patterns and writing them, but it's reasonable. And once written, they're easily applied elsewhere."

Both participants indicated that the sketch language was far easier to work with than regular expressions; the first participant pointedly commented that "Regular expressions [are not sufficiently human-usable]."[5]

[5]We have suppressed his more "colourful" phrasing.

5.4.3 Is the tool sufficiently accurate?

Both participants indicated that the tool was reasonably accurate: the first study participant thought he had detected a false negative, but on further investigation it turned out the tool was correct. The second participant noted there were a few false positives, but he did not notice any false negatives outside those cases he decided were not worth capturing.

Both participants felt the tool was useful for polylingual dependency analysis, especially because they were not aware of any other alternatives, but would not trust the predictions without some manual investigation of the results. In its current state though, it would dramatically reduce the number of false positives they normally deal with in such investigations, saving considerable time. The participants both suggested some additional constructs to improve the expressiveness of the pattern language, including the ability to combine patterns using set operations (e.g., intersection).

5.4.4 Does the approach work?

How the developers configured the tool varied: the first participant tended to write highly tailored and precise patterns which attempted to match dependency features as specifically as possible, while the second participant was more flexible in his matches and more willing to ignore cases which were deemed to be too much work to catch, or to ignore refining existing patterns if they were close enough. Their attitudes towards configuring the tool reflected their tasks, and the tool seemed amenable to both approaches.

6. DISCUSSION AND FUTURE WORK

Qualitative evaluation of accuracy and effort. Our evaluation of DSketch reports the participants' perceptions of the effort required to configure the tool, and of the tool's prediction accuracy for their case study task. As our studies were exploratory, we do not quantitatively report information retrieval metrics, nor metrics of effort. Our tooling sufficed to determine strengths and weaknesses of the approach; only after these results are taken into account in the tool design will a formal and quantitative experiment be appropriate [18].

Improving pattern expressiveness. The study participants identified a few additional notations or capabilities to allow writing more expressive or specific patterns. Some of these limitations, specifically the lack of regular expression closure support, were due to limitations we encountered when trying to use back-referencing in our patterns to extract matches in these cases (only the last back-referenced match in such patterns are captured). Resolving these issues may require adopting a different technology for applying the patterns, or creating specialized support for these cases.

Usability and performance. Both participants suggested modifications to the interface to adapt it more succinctly to their desired workflow: the first participant wished for more extensive pattern debugging capabilities, the second expressed a strong desire to see the tool even more tightly integrated into the Eclipse IDE such that it would resemble the existing Java tooling support in operation and usage; both indicated that it would be desirable to support their tendency to choose a particular code snippet as exemplar of the pattern they are trying to write. Performance is also an issue: on the JPetStore system, the second participant noted that the tool paused notably during its analysis. Since

his preferred workflow involved iteratively refining and applying patterns to the system, it was important to him that the speed of the tool be as close to real-time as possible.

Adapting to large-scale system issues. As the systems under analysis by DSketch grow in size, we expect (and our participants expressed similar concerns) that it will become more difficult to keep track of which patterns are contributing effectively to the tool configuration. The likelihood of the tool's confusing dependency matches due to its simplified notation language and heuristics will increase, leading to more false positives in the developer results. Rather than put additional onus on the developer to supply increasingly detailed configuration information, we would like to explore mechanisms that could allow the tool to adapt itself and its heuristics to the systems it analyzes, perhaps by leveraging feedback from the developer as to the accuracy of its predictions as a means of learning, or by providing the developer feedback as to which of their patterns (and why) are proving effective or ineffective in their investigation of the system. We are examining the feasibility of such extensions.

7. CONCLUSION

DSketch provides a lightweight approximation of semantically-aware tool support for polylingual dependency analysis. Developers configure the tool using simple pattern specifications that are easy to write, and provide reasonable approximations of the dependency syntax in their system. DSketch leverages these patterns to extract identifiers from the developer's system and predict where polylingual dependencies exist. The developer can iterate on their patterns to refine the set of dependencies until satisfied with the results, and can share these patterns with other developers for work in the same, or similar systems.

We conducted case studies with industrial developers to understand the strengths and weaknesses of our approach. Our participants were able to configure DSketch reasonably well with a brief period of training for polylingual dependency analysis, and were able to adapt the configuration of the tool to new situations easily. Both participants successfully used the tool to detect dependencies in their systems, and were satisfied with the accuracy of its predictions.

Some further details of the DSketch project can be found at `http://lsmr.cs.ucalgary.ca/project/dsketch`.

8. ACKNOWLEDGMENTS

We thank Soha Makady, Rylan Cottrell, and the anonymous reviewers for their feedback. This work was supported by the Natural Sciences and Engineering Research Council of Canada through a Discovery Grant and a Postgraduate Scholarship and by the Alberta Informatics Circle of Research Excellence through a Graduate Scholarship.

9. REFERENCES

[1] R. Arnold and S. Bohner. Impact analysis: Towards a framework for comparison. In *Proc. Conf. Softw. Maintenance*, pp. 292–301, 1993.

[2] D. Atkinson and W. Griswold. Effective pattern matching of source code using abstract syntax patterns. *Softw. Pract. Exper.*, 36(4):413–447, 2006.

[3] T. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proc. Working Conf. Reverse Eng.*, pp. 102–109, 1994.

[4] S. Bourne. *An Introduction to the UNIX Shell.* Bell Laboratories, 1977.

[5] S. Burson, G. Kotik, and L. Markosian. A program transformation approach to automating software re-engineering. In *Proc. IEEE Int. Comput. Softw. Appl. Conf.*, pp. 314–322, 1990.

[6] B. Cossette and R. Walker. Polylingual dependency analysis using island grammars: A cost versus accuracy evaluation. In *Proc. IEEE Int. Conf. Softw. Maintenance*, pp. 214–223, 2007.

[7] M. Furr and J. Foster. Polymoprhic type inference for the JNI. In *Proc. Europ. Symp. Progr.*, pp. 309–324, 2006.

[8] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proc. Int. Conf. Softw. Eng.*, pp. 232–242, 2009.

[9] R. Holmes and R. Walker. Supporting task-specific source code dependency investigation. In *Proc. IEEE Int. Wkshp. Visualiz. Softw. Understand. Analys.*, pp. 100–108, 2007.

[10] K. Kontogiannis, P. Linos, and K. Wong. Comprehension and maintenance of large-scale multi-language software applications. In *Proc. Int. Conf. Softw. Maintenance*, pp. 497–500, 2006.

[11] R. Koppler. A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6):637–649, 1997.

[12] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[13] D. Moise and K. Wong. Extracting and representing cross-language dependencies in diverse software systems. In *Proc. Working Conf. Reverse Eng.*, pp. 209–218, 2005.

[14] L. Moonen. Generating robust parsers using island grammars. In *Proc. Working Conf. Reverse Eng.*, pp. 13–24, 2001.

[15] L. Moonen. Lightweight impact analysis using island grammars. In *Proc. Int. Wkshp. Progr. Comprehension*, pp. 219–228, 2002.

[16] M. Moriconi and T. Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Trans. Softw. Eng.*, 16(9):980–992, 1990.

[17] G. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[18] G. Murphy, R. Walker, and E. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Trans. Softw. Eng.*, 25(4):438–455, 1999.

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

[20] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Syst. J.*, 13(2):231–256, 1974.