Understanding Software Application Interfaces via String Analysis

Evan Martin*
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
eemartin@csc.ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

ABSTRACT

In software systems, different software applications often interact with each other through specific interfaces by exchanging data in string format. For example, web services interact with each other through XML strings. Database applications interact with a database through strings of SQL statements. Sometimes these interfaces between different software applications are complex and distributed. For example, a table in a database can be accessed by multiple methods in a database application and a single method can access multiple tables. In this paper, we propose an approach to understanding software application interfaces through string analysis. The approach first performs a static analysis of source code to identify interaction points (in the form of interface-method-call sites). We then leverage existing string analysis tools to collect all possible string data that can be sent through these different interaction points. Then we manipulate collected string data by grouping similar data together. For example, we group together all collected SQL statements that access the same table. Then we associate various parts of aggregated data with interaction points in order to show the connections between entities from interacting applications. Our preliminary results show that the approach can help us understand the characteristics of interactions between database applications and databases. We also identify some challenges in this approach for our future work.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques — *modules and interfaces*

General Terms

Design, Experimentation, Verification

Keywords

program understanding, database-driven applications

1. INTRODUCTION

Today's applications often interact with a variety of independent external components such as various pieces of hardware, both local and remote software applications, as well as ubiquitous databases. In a services-oriented architecture (SOA) [6], such as web services, independent remote software applications cooperate and communicate with one another via some standardized protocol in the form of XML string data. There is also a rapid increase in the digitization of information being accompanied by exponential increases in networking, storage, and computing capacities [1]. Database systems, as the de facto standard for the management and efficient access of persistent data, have also become increasingly popular. As such, database systems have become the foundation for applications that require access to persistent data. Although different, each of these application domains shares a common thread, namely they interface with an external component in some standardized way. Furthermore, these interfaces primarily pass character streams such as XML for web-services and SQL for database-driven applications.

Along with the growing popularity of these types of systems, the interfaces between interacting software applications in these systems have become increasingly complicated and distributed. Therefore, developers are faced with the challenge of understanding and maintaining these interfaces. To address this challenge, we propose a new approach that leverages the existing static string analysis tools [4,8] and further analyzes their generated results to provide valuable information that may be used not only to verify communication protocol compliance but also to increase our understanding of software application interfaces. This increased understanding can serve several functions such as test case generation, development and debugging tasks, as well as development of design choices that lead to performance gains.

In our new approach, we first perform a static analysis of the source code to identify all possible interaction points. For example, we can search for JDBC-method-call sites since they have a specific method signature whose arguments are SQL statements. Subsequently we use the existing static string analysis tools [4, 8] to collect all possible string data (in the form of automata) that can be produced at these interaction points. Then we group together collected string data that interact with the same entities in the interacting software application. For example, we group together collected SQL statement strings that access the same table in a database. We then construct associations between parts of the aggregated string data and the corresponding interaction points in the source code so that the connections between external entities, such as a database table, and specific internal entities, such as methods in the source code, can be understood. We have implemented a tool for the approach and applied the tool on several database applications. The preliminary results show that our approach can help understand application interfaces with an aggregated-interface view between database applications and databases. This aggregated SQL statements with method-call annotations provide guidance on

*Supported in part by an IBM Ph.D. Fellowship.

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

understanding the connections of interacting applications. At the same time, we also identify several challenges for our future work in extending this approach.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes our approach. Section 4 shows our preliminary results of applying our approach. Section 5 concludes the paper with future directions.

2. RELATED WORK

Our current implementation focuses on interfaces that primarily transport character streams and leverages the Java String Analysis (JSA) library [4] to collect string data at specific call sites. JSA develops a mechanism for generating models of Java strings. In particular, JSA statically performs a conservative string analysis of an application and creates automata that express all the possible values a specific string can have at a given point in the application. Several other subsequent tools have been built based on JSA. Based on JSA, Gould et al. [7] developed the JDBC-Checker tool that statically checks the type correctness of dynamically-generated SQL queries. Halfond and Orso [8] developed the Amnesia tool that statically constructs a model of the legitimate queries that could be generated by the application, and dynamically checks runtime-generated queries against the model. The tool is used for detecting and preventing SQL injection attacks at run-time. All these existing approaches based on JSA focus on analysis of individual call sites, whereas our approach focuses on collective information gleaned from multiple call sites.

There exist a number of approaches for extracting sequencing constraints among method calls in a component interface. Ammons et al. [2] developed the mining specification approach for extracting a probabilistic finite state automaton from C program traces. Whaley et al. [16] developed approaches to extract Java component interfaces in the form of multiple finite state automata based on static analysis and dynamic analysis. Xie and Notkin [19, 20] developed two techniques for extracting abstract-object-state machines from the execution of unit tests for a component interface. In finite state automata extracted by these existing approaches, states are object states and transitions are method calls in the component interface, whereas in our new approach, transitions in string automata are string values. In addition, these existing approaches are often dynamic analysis techniques that work on a single component interface such as public methods of a class, whereas our new approach is a static analysis technique that focuses on interfaces of software applications.

There exist several approaches for testing database applications. Zhang et al. [21] developed a tool that generates database instances as inputs for testing database applications. Chays et al. [3] developed a set of tools called AGENDA to test relational database applications. Kapfhammer and Soffa [9] defined a family of test adequacy criteria for database-driven applications. The criteria use dataflow information associated with entities in a database. Suarez-Cabal and Tuya [15] also defined an SQL coverage measurement for testing database applications. These existing approaches focus on testing database applications whereas our new approach focuses on understanding software application interfaces such as database application interfaces.

3. APPROACH

Leveraging existing tools such as the Byte Code Engineering Library (BCEL) [5], JSA [4], and Amnesia [8], we have developed a tool that statically scans Java byte code to derive a mapping that associates entities from two interacting applications, such as a mapping between specific sections of application code with specific database objects. In doing so, one may identify various sections of code that may not only implement different functionalities but reside in different parts of the code base yet still use another application such as a database in similar ways. In short, our technique allows us to find similarities between sections of code not based on call or type hierarchies but on how they interact with another application such as a database. The remainder of this section outlines the steps necessary to derive these mappings.

```
public ResultSet executeQuery(String sql) {...}
public int executeUpdate(String sql) {...}
public boolean execute(String sql) {...}
```

Figure 1: JDBC method calls used as hotspots.

Figure 2: An example SQL-query model.

3.1 Interaction-Point Identification

Two interacting applications interact with each other through the interface between them. Our approach first identifies interaction points between two interacting applications called *hotspots*. When we focus on interactions of a database application and a database, hotspots are specific method calls in the code that issue SQL queries to the database. In Java, the methods that allow the application to query the database are specific method calls in the JDBC[1] API shown in Figure 1. Our tool uses BCEL [5] to scan the byte code to identify all JDBC method calls in the application as hotspots. However, queries constructed at different program locations are sometimes passed as the argument of an application method (e.g., `Database.query(String queryStr)`), which then invokes JDBC method calls. To handle this situation, we construct call chains [13] for each hotspot so that the tool user can specify n-level callers of hotspots as the interaction points (n is 0 by default), which are inputs to the JSA [4] library for string automaton generation described below.

3.2 String-Automaton Generation

The next step constructs automata for each caller of the JDBC methods. This set of automata represents all possible SQL queries the application may perform on the database. The JSA [4] library provides character level automata for any method that accepts a `String` argument. More specifically their technique creates a flow graph that represents the control flow of the program and subsequently creates a Non-Deterministic Finite Automaton (NDFA) from the control flow graph that expresses all possible values the `String` argument can take. Amnesia [8] further refines the results by transforming this character level automaton into a token level automaton. Amnesia performs a depth first traversal of the character level automaton identifying SQL keywords, operations, or literal values as tokens and creates a new automaton that uses these tokens as the transitions. For example, a character level sequence of transitions with the characters 'F', 'R', 'O', and 'M' is recognized as the SQL `FROM` keyword and replaces this set of transitions with a single transition labeled FROM. The result is an NDFA whose transitions are SQL keywords, operations, or database object identifiers. Additionally the flag `VAR` is used to indicate user input that can not be resolved via static analysis. For example, the SQL-query model shown in Figure 2 is produced

[1] `http://java.sun.com/products/jdbc/`

for the following method call: `executeQuery("SELECT Level FROM CustomerPriority")`

3.3 String-Automaton Grouping

By performing a depth first traversal of the SQL-query model produced by Amnesia [8], we can group the automata for each hotspot according to any number of desired characteristics. For example they may be grouped based on SQL keywords or operations, by what database objects are accessed such as specific tables or columns, by what sections of the code the automata generated from, or some combination of each. In our current implementation we group the automata for each hotspot according to what tables they access in the database. This grouping in conjunction with the call chains constructed in Section 3.1 provides the desired associations between application code and database objects. For example, given a query `SELECT * FROM Table1, Table2`, we visit the nodes of the automaton and extract the set of tables that the query is performed on. In this example, the set includes `Table1` and `Table2`. We only group those queries in which the set of tables are equivalent. Once the desired grouping is performed we perform a union operation followed by a minimization operation to reduce the grouped automaton into a single automata. Subsequently we traverse all paths of the reduced automata to label each state with a set of interaction points constructed in Section 3.1.

4. PRELIMINARY RESULTS

In this section we present preliminary results of the proposed approach to analyzing Java database-driven applications to reveal associations between the application code and the database objects by analyzing all possible SQL queries that may be issued to the database.

4.1 Subjects

Our initial studies were conducted on four publicly available database-driven applications. Three of these are web applications that accept user inputs through a web form and use the inputs to build queries issued to an underlying database. These applications include Employee Directory, Bookstore, and Classifieds and are available at GotoCode (`http://www.gotocode.com`). The fourth application, ReBaTe, is an open source project available from SourceForge (`http://sourceforge.net/projects/rebate/`). ReBaTe is not a web application but a stand-alone Java application that uses a Swing interface to accept user inputs and build queries issued to an underlying database.

Table 1 provides some basic information about each of the subjects. Specifically for each subject the table indicates the name (*Subject*), a brief description (*Description*), code size in units of lines of code (*LOC*), and number of identified hotspots (*Hotspots*).

Table 1: Experimental subjects

Subject	Description	LOC	Hotspots
Bookstore	Online bookstore	16959	71
Classifieds	Online classifieds system	10949	34
EmployeeDir	Online employee directory	5658	23
ReBaTe	Requirements-based testing tool	6036	27

4.2 Results

Figure 3 shows the union and subsequent minimization of four automata for the ReBaTe application, corresponding to four different hotspots that access the same table, namely the `Defect` table. The transitions are labeled with tokens corresponding to SQL keywords, operators, or database object identifiers such as table names or attribute names. The transitions labeled with the keyword `VAR` signify a variable and most often correspond with user inputs. The initial states are marked with an empty arrow and all terminating states are indicated by double-circles. Finally, each state contains a set of numbers. This set of numbers corresponds to each hotspot that traverses this state. Recall that Figure 3 is the union of four automata each corresponding to a different hotspot. In this example, each of the four hotspots is located in the singleton class `edu.ncsu.port.db.Database` in various locations as indicated in Figure 4. The methods in Figure 4 each issue an SQL query to the database via a JDBC method but more importantly each of these methods shares a similarity that is not immediately obvious from the source code, namely they each only access the `Defect` table. The numbers 1, 2, 3, and 4 in Figure 3 correspond to the methods at Lines 150, 252, 325, and 773, respectively (shown in Figure 4). This labeling illustrates what parts of the automata are contributed by each hotspot and exactly what similarities these hotspots share in terms of possible string data that they produce.

By inspecting the automaton shown in Figure 4, we found that the queries generated at Locations 3 and 4 are in fact semantically equivalent: the only syntactic difference is that "`TestCaseId`" is preceded by "`Defect.`" in Location 3 but is not in Location 4. The observation indicates that the two methods corresponding to Locations 3 and 4 are redundant and they should be merged into a single method. We also observed that Locations 1 and 2 construct similar queries for `ProductId` and `Id`, respectively. The preceding useful information about the interface cannot be easily attained by looking at the source code without using our tool.

We can view database accesses as concerns [10] that crosscut the whole database application. Furthermore, database accesses related to the same table represent a finer-granularity of concerns. In a high-level view, we can consider our aggregated automata shown in Figure 3 as a type of concern graphs proposed by Robillard and Murphy [12]. But note that our approach does not require extra user inputs for constructing this type of concern graphs. Although a database application could be large, the total number of tables accessed by a database application could be relatively small. Therefore, it could be feasible to first use the grouped automata produced by our approach to understand a database application.

For other hotspots in ReBaTe or in other subjects, we often found that grouped automata were useful for understanding the database application interfaces. But on the other hand, we also encountered two contrary situations: we could not find a substantial number of groupings or a grouped automaton was too complicated for understanding.

5. CONCLUSION

We have developed a new approach that helps understand software application interfaces such as database application interfaces. In this approach, we leverage the existing static string analyzers [4, 8] to collect possible string data of SQL statements at specified call sites. Then we group together the SQL statements that access the same table. We produce an aggregated view of SQL statements where corresponding call sites are annotated. We have developed a tool to support the approach and applied the tool on several database applications. The preliminary results show that the extracted views can often help understand database application interfaces.

We plan to pursue several future directions for our new approach. First, we plan to explore advanced techniques (such as data mining [18], visualization [14], and concept analysis [17]) to make in-

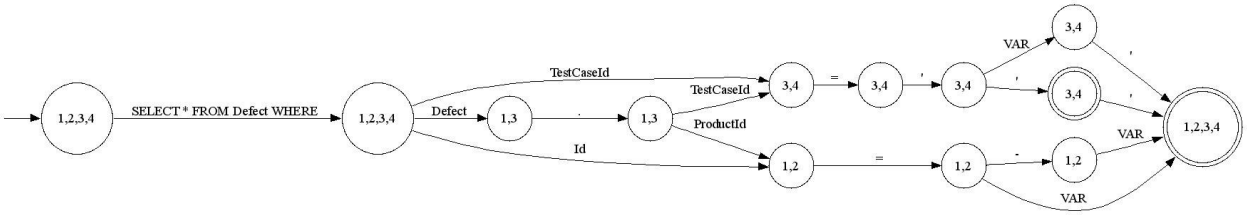

Figure 3: The union of four automata accessing the `Defect` table with a `Select` statement.

```
1  public class Database {
...
150 public static Collection getDefTab(int prodId) {
          ...
          rs = Database.query(sql);
          ...
    }
...
252 public static DefectDataModel getDef(int defId) {
          ...
          rs = Database.query(sql);
          ...
    }
...
325 public static Collection getDefForTC(String tcId) {
          ...
          rs = Database.query(sql);
          ...
    }
...
773 public static Collection getDefTabByTC(String tcId) {
          ...
          rs = Database.query(sql);
          ...
    }
```

Figure 4: The call hierarchy corresponding to the automaton in Figure 3.

formation in grouped automata more understandable, because our straightforward grouping sometimes produces a complicated combined automata. In addition, we plan to explore different strategies of grouping other than the simple approach currently taken, because sometimes we do not have a substantial number of groupings. Second, we plan to combine our approach with dynamic analysis to help testing, debugging, and performance problem diagnosis of database applications. Third, we plan to apply the approach to other types of software application interfaces that involve character streams in Java, such as reflection, Remote Method Invocation (RMI), XML, Java resource bundles, and web services. Finally, we plan to apply our approach in system performance optimization. For example, given a set of queries and a database, view selection returns definitions of views that, when materialized in the database, would reduce the evaluation costs of the queries. Optimizing the layout of stored data using view selection has significant performance implications on the entire database system. Unfortunately the optimization problem is intractable. Li et al. [11] use an integer-programming model to obtain optimal solutions to the problem of view selection for aggregate queries on databases. This model in conjunction with our approach can provide an automated technique to realize performance gains for database-driven applications simply by defining the necessary views and instrumenting the code to execute queries on those views rather than the underlying tables.

Acknowledgments

We would like to thank Alex Orso and William G.J. Halfond for providing their AMNESIA tool and sharing their experimental subjects for our use in the work described in this paper.

6. REFERENCES

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proc. 28th International Conference on Very Large Databases*, 2002.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[3] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Softw. Test., Verif. Reliab.*, 14(1):17–44, 2004.

[4] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[5] M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

[6] T. Erl. *Service-Oriented Architecture : A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, 2004.

[7] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. 26th International Conference on Software Engineering*, pages 645–654, May 2004.

[8] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proc. IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, November 2005.

[9] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–107, 2003.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[11] J. Li, Z. A. Talebi, R. Chirkova, and Y. Fathi. A formal model for the problem of view selection for aggregate queries. In *Proc. 9th East European Conference on Advances in Databases and Information Systems*, pages 125–138, September 2005.

[12] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. 24th International Conference on Software Engineering*, pages 406–416, 2002.

[13] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11, 2004.

[14] J. T. Stasko, J. B. Domingue, M. H. Brown, and B. A. Price. *Software Visualization*. MIT Press, 1998.

[15] M. J. Suarez-Cabal and J. Tuya. Using an SQL coverage measurement for testing database applications. In *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2004.

[16] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[17] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. *Ordered Sets, Ivan Rival Ed., NATO Advanced Study Institute*, 83:445–470, September 1981.

[18] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[19] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, pages 290–305, Nov. 2004.

[20] T. Xie and D. Notkin. Automatic extraction of sliced object state machines for component interfaces. In *Proc. 3rd Workshop on Specification and Verification of Component-Based Systems*, pages 39–46, October 2004.

[21] J. Zhang, C. Xu, and S. C. Cheung. Automatic generation of database instances for white-box testing. In *Proc. 25th International Computer Software and Applications Conference.*, pages 161–165, 2001.