Towards a Method for Reducing the Test Suites of Database Applications

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
gkapfham@allegheny.edu

***Abstract*—Database applications are commonly implemented and used in both industry and academia. These complex and rapidly evolving applications often have frequent changes in the source code of the program and the state and structure of the database. This paper describes and empirically evaluates a test suite reduction technique that improves the efficiency of regression testing for database applications by removing redundant tests. The experimental results show that the reduced test suites are between 30% and 80% smaller than the original test suite, thus enabling a decrease in testing time ranging from 7% to 78%. These empirical outcomes suggest that test suite reduction is a viable method for controlling the costs associated with testing rapidly-evolving database applications.**

***Keywords*-database application; test suite reduction**

I. Introduction

The database is a critical component of many real-world software applications that are used in industry and academia. Database applications have been implemented to create electronic journals [1], scientific data repositories [2], and electronic-commerce applications [3]. Indeed, Silberschatz et al. observe that "practically all use of databases occurs from within application programs" [4, pg. 311]. Moreover, many database applications rapidly evolve as software developers change the source code, database administrators modify the database's schema, and external application programs add data to and delete data from the database.

As developers and administrators change a database application, they may introduce faults into the program's source code or the database's state or structure. The goal of regression testing is to ensure that these changes did not introduce faults into the existing functionality by running a test suite $T = \langle t_1, t_2, \ldots, t_n \rangle$ [5]. Yet, as the number of test cases in the test suite grows and the database's state increases in size and complexity, re-executing the test suite becomes more and more expensive. By removing redundant tests from the test suite, reduction decreases test execution time, thereby making regression testing faster [5].

This paper presents a test suite reduction method specifically tailored for database applications. First, this database-aware approach collects per-test case requirements that capture how the program interacts with a relational database during testing [6]. Next, it uses an overlap-aware greedy algorithm to identify which test cases are redundant and removes them from the test suite [7]. Even though it is normally smaller than the original test suite, the reduced test suite, denoted T', is guaranteed to cover the same requirements as T. Test suite T' often executes faster than T both because it has fewer tests and because it no longer runs the costly operations associated with starting up and shutting down the database for the discarded tests. Until the database application undergoes major changes, the efficiency of regression testing can be improved by using T'.

II. Database-Aware Technique

Similar to [3], this paper focuses on the regression testing of database applications that are written in the Java programming language and interact with a relational database management system (RDBMS) by using a Java Database Connectivity (JDBC) driver to submit structured query language (SQL) strings. Both the test coverage monitor and the regression testing technique interoperate with any type of relational database management system that supports JDBC [6]. Moreover, the coverage monitor captures database interactions at all of the relevant levels of interaction granularity: database, relation, attribute, record, and attribute value [6]. Intuitively, a database-aware requirement for test case $t_i \in T$ corresponds to the series of method calls leading up to a database interaction point and the set of relational database entities that were defined and used during testing. A report that saves database interactions in a fine-grained manner (e.g., the record or attribute value level) provides more context for debugging than one that saves coverage in a coarse-grained fashion (e.g., the database or relation level).

The test suite reducer employs an overlap-aware greedy algorithm that is based on the approximation algorithm for the minimal set cover problem [7]. Greedy reduction with overlap awareness iteratively selects the most cost-effective test case for inclusion in the reduced test suite T'. During every successive iteration, the overlap-aware greedy algorithm re-calculates the cost-effectiveness for each leftover test according to how well it covers the remaining test requirements. The reduction technique terminates when T' covers all of the test requirements that the initial tests cover. For instance, if tests t_i and t_j both interact with the fifth record of relation r, then the reducer will discard the one with the greater execution time. However, if test case t_k efficiently covers the first ten records of relation r, then the greedy algorithm would keep it and discard both t_i and t_j.

Database Application - $\|T\|$	**Relation**	**Attribute**	**Record**	**Attribute Value**	**Avg. of Granularities**
Reminder (RM) - 13	$(7, .46)$	$(7, .46)$	$(10, .30)$	$(9, .31)$	$(8.25, .37)$
FileFind (FF) - 16	$(7, .56)$	$(7, .56)$	$(11, .31)$	$(11, .31)$	$(9, .44)$
Pithy (PI) - 15	$(6, .60)$	$(6, .60)$	$(8, .70)$	$(7, .53)$	$(6.75, .55)$
StudentTracker (ST) - 25	$(5, .80)$	$(5, .76)$	$(11, .56)$	$(10, .60)$	$(7.75, .69)$
TransactionManager (TM) - 27	$(14, .48)$	$(14, .48)$	$(15, .45)$	$(14, .48)$	$(14.25, .47)$
GradeBook (GB) - 51	$(33, .35)$	$(33, .35)$	$(33, .35)$	$(32, .37)$	$(32.75, .36)$
Avg. of Applications - 24.5	$(12, .51)$	$(12.17, .50)$	$(14.67, .40)$	$(13.83, .44)$	

Figure 1. Reduction in Test Suite Size for the Database Applications — $(|T|, \text{RFFS}(T, T'))$ for All Data Points.

III. Empirical Study

The empirical study used six database applications that range in size from 548 to 1455 non-commented source statements (NCSS) [6]. These case study applications employ a wide variety of strategies to test the program's interaction with a relational database that contains up to nine relations. The test suite for each case study application uses the DBUnit 2.1 extension of JUnit 3.8.1. Every application interacts with an HSQLDB in-memory relational database; since the applications use JDBC, it is possible to configure them to use other databases such as PostgreSQL or MySQL. Figure 1 shows that each application's test suite ranges in size from 13 to 51 test cases. Even though these suites are small and normally execute in seconds, they serve as a useful step towards empirically evaluating database-aware test suite reduction. Also, these small test suites can consume a considerable amount of execution time if they are configured to inspect a lot of the database's state and/or create test isolation by regularly restarting the database server.

To evaluate the effectiveness of the reduced test suites, we calculate two "higher is better" metrics: the reduction factor for size, or RFFS$(T, T') = (|T| - |T'|) \div |T|$ and the reduction factor for time, or RFFT$(T, T') = (\mathit{time}(T) - \mathit{time}(T')) \div \mathit{time}(T)$. To study the efficiency of reduction we measured execution time. Since a reducer never took more than one second for any database application or interaction level, this paper does not further consider efficiency.

Figure 1 gives the value of RFFS(T, T') for the six case study applications (due to space constraints, we later comment on the values for RFFS(T, T') instead of providing the full table). We observe that ST exhibits the best reduction factor (.69 on average) and GB has the worst (.36 on average) — a result that we ascribe to the fact that ST has great overlap in the requirements and GB has very little. Across all of the applications, the average value of RFFS was .51 at the relation level and .44 at the attribute value level, suggesting that noticeable reductions are often possible.

Since the chosen database applications only interact with a few relations, the results in Figure 1 reveal that the value of RFFS is normally the highest at the relation level and the lowest at the record level. If regression testing occurs at the relation level, then there is a substantial amount of overlap in requirement coverage, thus leading to high RFFS values. The results show that it may be advisable to reduce at the attribute level instead of the relation level because the (i) reduction algorithm is efficient at both of these levels, (ii) value of RFFS is comparable at these two levels, and (iii) attribute-level coverage report is often better at more fully capturing the behavior of the test cases [6].

Further analysis of the data in Figure 1 reveals an average decrease in RFFS at the transition from the attribute to the record level. Across all of the applications, we see that RFFS drops from .50 to .40 when the reducer analyzes at the record level instead of the attribute level, suggesting that the potential for reduction is limited if the tests often place different records into the database. Yet, Figure 1 shows that the average value of RFFS climbs to .44 from .40 when using requirements based on attribute values rather than records. When run at the attribute-value level, the reducer can better identify the overlap in requirement coverage because the tests often insert records that only differ by a few attribute values. Since the reducer is efficient at both the record and attribute-value levels and more effective at the finer level of granularity, it is sensible to reduce with requirements based on attribute values instead of records. Overall, these high RFFS values lead to RFFT scores that range between .07 and .78, thus highlighting the value in further developing and investigating database-aware test suite reduction. As such, future work will investigate the fault-detection effectiveness of T and T' and extend the empirical study to consider new and different types of database applications.

References

[1] G. Marchionini and H. Maurer, "The roles of digital libraries in teaching and learning," *CACM*, 1995.

[2] R. Moore, T. A. Prince, and M. Ellisman, "Data-intensive computing and digital libraries," *CACM*, 1998.

[3] W. G. J. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter SQL-injection attacks," in *Proc of the 3rd WODA*, 2005.

[4] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 5th ed., 2006.

[5] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *TOSEM*, 1993.

[6] G. M. Kapfhammer and M. L. Soffa, "Database-aware test coverage monitoring," in *Proc of the 1st ISEC*, 2008.

[7] A. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, "Test suite reduction and prioritization with call trees," in *Proc. of 22nd ASE*, 2007.