Database-Aware Test Coverage Monitoring

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

Mary Lou Soffa
Department of Computer Science
University of Virginia
soffa@cs.virginia.edu

ABSTRACT

Unlike traditional programs, a database-centric application interacts with a database that has a complex state and structure. Even though the database is an important component of modern software, there are few tools to support the testing of database-centric applications. This paper presents a test coverage monitoring technique that tracks a program's definition and use of database entities during test suite execution. The paper also describes instrumentation probes that construct a coverage tree that records how the program and the tests cover the database. We conducted experiments to measure the costs that are associated with (i) instrumenting the program and the tests and (ii) monitoring coverage. For all of the applications, the experiments demonstrate that the instrumentation mechanism incurs an acceptable time overhead. While the use of statically inserted probes may increase the size of an application, this approach enables database-aware coverage monitoring that increases testing time from 13% to no more than 54%.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Verification, Experimentation

Keywords: database application, test coverage monitoring

1. INTRODUCTION

The database is a critical component of many modern software applications. However, a database is useful for storing information only if users can correctly and efficiently (i) query, update, and remove existing data and (ii) insert new data. To this end, software developers often implement database-centric applications that interact with a database. Indeed, Silberschatz et al. observe that "practically all use of databases occurs from within application programs" [28, pg. 311]. Yet, a database-centric application is very different from a traditional software system because it interacts with a database that has a complex state and structure.

The goal of conventional test coverage monitoring (e.g., [19, 22, 30]) is to record the program elements that the tests cover. For example, traditional monitoring schemes track the coverage of the nodes and edges in a program's control flow graph (CFG) (i.e., code coverage) or definition-use associations involving the program variables (i.e., data flow coverage). These approaches are unsuitable for database-aware coverage monitoring because they do not record how a program covers the database as the test cases execute. If a database-aware coverage criterion requires the tests to cover all of the relations in the database, the monitor must be able to determine how well the test cases fulfill this obligation.

This paper presents a database-aware test coverage monitor (TCM) that uses instrumentation to determine how both the program and the tests execute structured query language (SQL) statements (e.g., **select**, **update**, **insert**, and **delete**) that define and use the database. Suppose that a program method submits a SQL **select** statement that performs a query about the database's state. In this circumstance, the coverage monitor intercepts and analyzes the records that match the **select** query. Our technique also finds out how the test cases inspect and change the database. For example, assume that a test case uses the SQL **delete** command to remove records from the database before it executes the method under test (i.e., the tests may purge the database so that the execution of one test does not impact the other test cases). Once it captures the state of the database before and after the execution of the **delete**, the monitor identifies the records that were removed.

Figure 1 depicts the process of database-aware test coverage monitoring. This diagram shows that the instrumentation phase accepts an adequacy criterion (e.g., *all-record-DUs* [12, 13]), a test suite, and the program under test. We instrument the program under test by placing *probes* at key locations within the CFG. The execution of the instrumented program and test suite yields the database-aware coverage results. The coverage report records the definition and use of relational database entities in the context of both the method and the test case that perform the database interaction. The database-aware coverage results can be leveraged to calculate the adequacy (i.e., the "quality" or "goodness") of either a single test case or the entire test suite.

Similar to [7, 10], we focus on the testing and analysis of database-centric applications that (i) are written in the Java programming language and (ii) interact with a relational database management system (RDBMS) by using a Java Database Connectivity (JDBC) driver to submit SQL strings. We designed the TCM component so that it interoperates with *any* type of RDBMS, Java virtual machine (JVM), and operating system (OS). The coverage monitor

ISEC'08, February 19-22, 2008, Hyderabad, India.

also captures database interactions at all of the relevant levels of interaction granularity: database, relation, attribute, record, and attribute value [12, 13].

We designed the instrumentation technique to *statically* introduce the probes *before* test suite execution, as depicted in Figure 1. Alternatively, the coverage monitor performs *dynamic* instrumentation by introducing the probes *during* testing. Unlike the majority of coverage tools (e.g., Clover [14], Jazz [19], and Emma [27]), we maintain the coverage results on a per-test case basis in order to support efficient regression testing [32]. The coverage report also enables the assessment of test adequacy according to a variety of database-aware criteria such as those that focus on data flow (e.g., *all-record-DUs*) [13] or run-time behavior (e.g., *call stack coverage*) [17].

Using six database-centric applications as case studies, we performed experiments to evaluate the performance of the components that perform instrumentation and test coverage monitoring (Figure 1 highlights these modules with a grey background). The experimental results reveal that the static instrumentation technique requires less than six seconds to insert coverage probes into a database-centric application. We found that the use of statically inserted probes lengthens testing time by 12.5% while dynamic instrumentation causes a 53% increase in test execution time. The empirical results also demonstrate that using static instrumentation to record coverage at the finest level of database interaction granularity never increases testing time by more than 54%.

The important contributions of this paper include:

1. A discussion of the challenges that are associated with database-aware test coverage monitoring (Sections 1 through 3).
2. The description of the probes and the test coverage monitoring trees that record coverage information in a database-aware fashion (Sections 4.1 through 4.4).
3. An aspect-oriented implementation of an instrumentation technique that efficiently generates and stores the database-aware coverage report (Sections 4.5 and 4.6).
4. An empirical evaluation of the costs that are associated with instrumenting the program and the test suite and monitoring coverage during testing (Section 5).

2. MOTIVATING EXAMPLE

We use the `FindFile` database-centric application to motivate the need for a database-aware test coverage monitor. `FindFile` stores meta-data about the files within a directory structure and it allows the user to search for files that match a provided signature.[1] For example, suppose that `FindFile`'s database was populated with information about all of the files in the `/usr/bin/` directory. If the command `java FindFile /usr/bin/bi` was executed on a GNU/Linux workstation, the `listFiles(String name)` operation would return a formatted string containing the file paths `/usr/bin/bibtex` and `/usr/bin/bison`.

`FindFile` interacts with a database that contains a single relation named *Files*. As shown in Figure 2, the *Files* relation contains two attributes called *Id* and *Path*. The *Files* relation stores records (e.g., $\langle 25,$ /usr/bin/bibtex$\rangle$) and attribute values (e.g., $\langle Id, 25\rangle$). Figure 2 shows the steps that occur during the testing of the `listFiles` method. In the first step, `testListFiles` inputs a file signature (e.g., `/usr/bin/bi`) to the method under test. This method uses the input signature to construct the SQL **select** statement in Figure 2. During the second step, `listFiles` uses the JDBC interface to submit the **select** to the database.[2] After executing the **select** query, the third step requires the RDBMS to return a result set that contains the two matching records. In the fourth step, `listFiles` analyzes the result set and places all of the file paths into a string that is returned to the test case for further analysis.

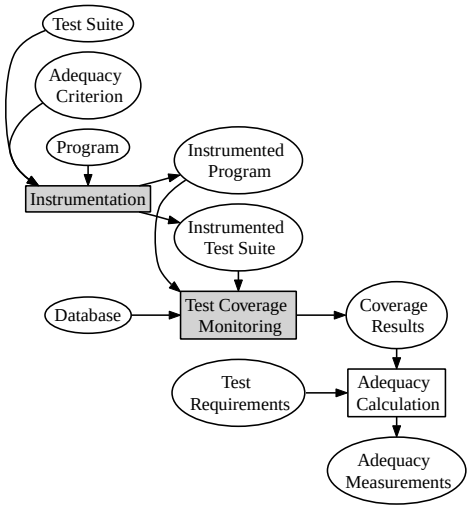

Figure 1: Test Coverage Monitoring Process.

A traditional coverage monitor would reveal that `testListFiles` invoked the `listFiles` method with the previously described inputs and outputs. This conventional report may also include information about the statements, branches, and def-use associations within `listFiles` that the test covered during execution. Coverage results that are not database-aware do not reveal how the method used the state (i.e., records and attributes values) and structure (i.e., relations and attributes) of `FindFile`'s database. Moreover, the coverage monitor cannot extract this type of information through traditional means such as examining the output of the `listFiles` method. This is due to the fact that the method under test may improperly process the result set and thus create an incorrect return value. Figure 2 also provides an **insert** command that `FindFile` may submit during testing. A standard coverage monitor is unable to determine how the **insert** changes the state of the database because it does not have access to the contents of the *Files* relation. Even though this example focuses on the SQL **select** and **insert** statements, we also encounter the same need for database-aware monitoring when the program employs the SQL **update** and **delete** commands.

There are also a wide variety of techniques that cannot be applied to the testing of `FindFile` without database-aware coverage results. For instance, without the details concerning a program's database interactions, it is not possible to

[1] `FindFile` is bundled with the in-memory Java database available from `http://www.hsqldb.org`. Section 5 provides additional details about `FindFile` since we use it as a subject during experimentation. We clarify the discussion in Section 2 by simplifying the description of `FindFile`'s methods and database.

[2] This query uses the **ucase** function to convert all of the characters in *Path* to uppercase. The **select** also uses the SQL wildcard character (i.e., %) to match any string of zero or more characters.

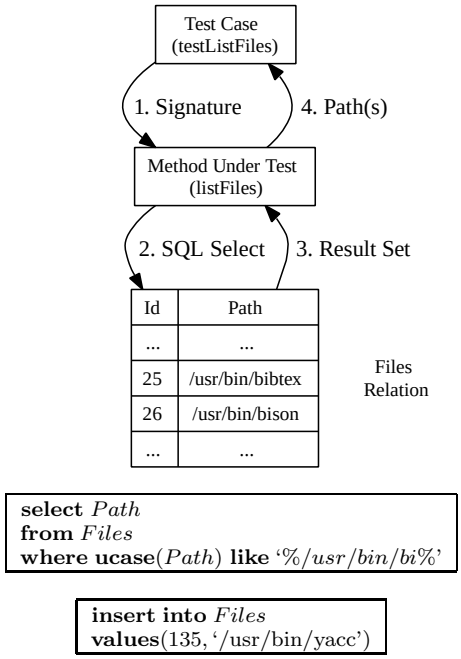

Figure 2: Test Coverage Monitoring Example.

calculate the adequacy of a test suite according to criterion such as *all-record-DUs* [12, 13]. A database-aware coverage monitor is also indispensable since many regression testing tasks (e.g., reducing or prioritizing the test suite) use coverage data. Techniques for database-aware debugging and automated fault localization also require information about how the program interacted with the database during test suite execution. In summary, all of these examples clearly motivate the need for a database-aware coverage monitor that identifies how the relational database entities were defined and used during test suite execution.

3. MONITORING CHALLENGES

3.1 Location of the Instrumentation

Figure 3 shows a program and a test suite that execute on a JVM and interact with a Java-based RDBMS (we classify the database manager's JVM as optional because some RDBMSs are native applications). Previously developed approaches to instrumentation place the probes in the (i) program under test [19], (ii) JDBC driver [3], (iii) RDBMS [6], or (iv) operating system (OS) [9]. However, the TCM component cannot place instrumentation into either the RDBMS or the OS because we want the technique to function properly for any combination of a JDBC-accessible database and a modern operating system. The JDBC Web site reveals that there are currently over two hundred different JDBC drivers that (i) are written in a combination of the Java, C, and C++ programming languages and (ii) vary in their internal structure and behavior [18]. Without a general and automated technique for instrumenting any JDBC driver, it is unrealistic for the coverage monitor to record database interactions by placing probes into all current drivers.

Most conventional coverage monitors insert the probes through the use of either a static approach or a dynamic JVM-based technique [19, 22, 30]. The wide variety of JVMs (e.g., the Sun HotSpot^{TM}JVM and the Jikes Research Virtual Machine (RVM) [1]) suggests that it is challenging to dynamically introduce the probes with the JVM that runs the test suite. In light of our analysis of the potential instrumentation locations, we implemented static and dynamic techniques that only modify the program and the test suite. Since a tester does not always have access to an application's source code, we designed the TCM component so that it can operate on Java source code, bytecode, or a combination of both types of program representations. Finally, the dynamic technique operates on well-established interfaces that exist in all JVMs (i.e., the current implementation performs *load-time* instrumentation using the JVM class loader).

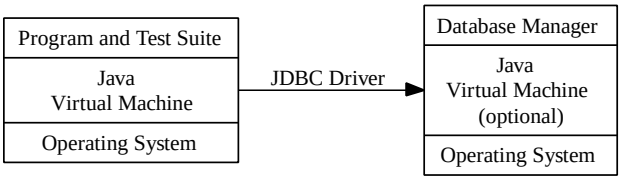

Figure 3: A Database-Centric Application.

3.2 Types of Instrumentation

Database-aware coverage monitoring is challenging because different RDBMSs support divergent dialects of SQL. For example, suppose that a database contains the relation $rel_j = \{t_1, \ldots, t_u\}$. Figure 4 provides two SQL **insert** commands that copy records t_{begin} through t_{end} from rel_j to $rel_{j'}$ whenever predicate Q holds. The Oracle **select** uses the **rownum** variable to bind the records in the **select**'s result set while the PostgreSQL statements use the **limit** and **offset** keywords. Our coverage monitor supports the testing of programs that use different RDBMSs because we restrict the parsing of the SQL strings to the syntactical elements that all dialects have in common. A probe determines how the **insert** modifies the database by first identifying $rel_{j'}$ with a regular expression and then extracting and differencing the before and after states of this relation.

Even when parsing of the SQL commands is not required, the probes must examine the state and structure of the database without inadvertently introducing defects into the program. Suppose that we are monitoring the coverage of the `listFiles` method in Figure 2. The probes intercept this result set and save the records that the **select** used. Yet, most JDBC drivers return a result set that only supports a single iteration from t_1 to t_u. If the probe iterates through the result set, then the instrumentation will incorrectly change the behavior of the program. The method under test will terminate if it attempts to analyze a result set that was exhausted by the test coverage monitor. Thus, a probe must preserve the correctness of the monitored application by transparently modifying the result set so that it supports multiple iterations.

4. DATABASE-AWARE COVERAGE

4.1 Database-Centric Applications

A database-centric application $\mathcal{A} = \langle P, \langle D_1, \ldots, D_n \rangle, \langle S_1, \ldots, S_n \rangle \rangle$ consists of a program P and databases $D_1, \ldots, D_n$ that are specified by relational schemas $S_1, \ldots, S_n$. A database D is a set of relations such that $D = \{rel_1, \ldots, rel_w\}$ and a relation $rel_j = \{t_1, \ldots, t_u\}$ is a set of records. For a relation with z attributes, each record is an ordered set

PostgreSQL

insert into $rel_{j'}$
  select $A_1, A_2, \ldots, A_z$
  from rel_j **limit** rec_{end} **offset** rec_{begin}
  where Q

Oracle

insert into $rel_{j'}$
  select $A_1, A_2, \ldots, A_z$ **from** rel_j
  where rownum $\geq rec_{begin}$
  and rownum $\leq rec_{end}$ **and** Q

Figure 4: Syntax for the SQL Statements.

of attribute values such that $t_k = \langle t_k[1], \ldots, t_k[z] \rangle$. The notation $t_k[l]$ denotes the value of the lth attribute of the kth record in a specified relation. We represent P's method m as a control flow graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$ where $\mathcal{N}$ and $\mathcal{E}$ are sets of nodes and edges, respectively.

Method m can contain one or more database interaction points, each of which corresponds to a node $N_r \in \mathcal{N}$. In the context of database-centric applications that use the JDBC interface, node N_r normally corresponds to the invocation of a method such as `executeQuery` (for the SQL **select** statement) or `executeUpdate` (for the SQL **update**, **insert**, and **delete** commands). Figure 5 reviews the general format of the SQL statements that m may submit to the database at N_r (as demonstrated in Figure 4 we handle properly nested SQL commands). Figure 5 also indicates that the SQL **select** statement is of *type* using while the **insert**'s type is defining. We classify the **update** and **delete** operations as type defining-using because they define relation rel_j and use any of the relations and attributes that are referenced by Q.

4.2 Overview of the Coverage Trees

The tree-based coverage report follows the format in Figure 6 and stores the definition and/or use of a database entity in the context of both the (i) current test case and (ii) method that performs the database interaction. We designed the coverage report to balance the benefits of fully capturing the execution context and database interactions with the time and space overheads associated with storing complete information. A tree that records coverage in a fine manner (e.g., the record or attribute value level) provides more context for debugging than a tree that saves coverage in a coarse fashion (e.g., the database or relation level). Yet, the creation and storage of a fine granularity tree normally incurs higher time overheads than the coarse tree.

Since database interactions occur via method calls, we construct a coverage tree with probes that operate *before* and *after* the execution of a method. Figure 7 compares the types of trees that these coverage probes can create. The test coverage monitor can construct either a *dynamic call tree* (DCT) or a *calling context tree* (CCT). Since the DCT contains a node for each method call, it preserves full testing context at the expense of having unbounded depth and breadth [2]. The DCT has unbounded depth because it fully represents recursion and it has unbounded breadth since it completely represents iterative method calls. Figure 7 indicates that the DCT incurs low TCM probe time overhead and moderate to high tree space overhead.

In contrast, the CCT has bounded depth because it coalesces nodes and uses back edges when methods are recursively invoked during testing. The CCT also has bounded breadth since it coalesces nodes when testing causes the iterative invocation of methods [2]. Figure 7 shows that the

select $A_1, A_2, \ldots, A_z$
from $rel_1, rel_2, \ldots, rel_w$
where Q

Type: *using*

(a)

delete from rel_j
where Q

Type: *defining-using*

(b)

insert into $rel_j(A_1, A_2, \ldots, A_z)$
values$(v_1, v_2, \ldots, v_z)$

Type: *defining*

(c)

update rel_j
set $A_l = F(A'_l)$
where Q

Type: *defining-using*

(d)

Q contains sub-predicates $V_\phi \; \Re \; V_\psi$
$\Re \in \{<, \leq, >, \geq, \neq, \textbf{in}, \textbf{between}, \textbf{like}\}$
$V_\phi \in \{A_1, A_2, \ldots, A_z\}$
$V_\psi \in \{\text{string}, \text{pattern}, \text{nested } \textbf{select}\}$

Figure 5: General Form of the SQL Operations.

CCT has low tree space overheads at the expense of slightly increasing the execution time of the probes. Since both the CCT and the DCT do not record a program's interaction with a relational database, they are not directly suited to maintaining the coverage of the tests for a database-centric application. To this end, we designed coverage trees (i.e., the DI-DCT and DI-CCT) that may incur additional time and space overhead because they are database-aware.

4.3 Traditional Trees

We use τ to denote any type of coverage tree (including those that are database-aware) and we use τ_{dct} and τ_{cct} to respectively stand for a DCT and a CCT. The following Definition 1 defines the dynamic call tree. The node $N_a \in \mathcal{N}_\tau$ is the *active* node that the probe references when it changes τ_{dct}. We use the notation $in(N_\rho)$ and $out(N_\rho)$ to respectively refer to the in-degree and out-degree of a node N_ρ. Definition 1 requires τ_{dct} to have a distinguished node N_0, the root, such that τ_{dct} does not contain any edges of the form (N_ϕ, N_0) (i.e., $in(N_0) = 0$). For all nodes $N_\phi \in \mathcal{N}_\tau - \{N_0\}$, Definition 1 requires $in(N_\phi) = 1$ (i.e., every node except the root must have a unique parent).

Definition 1. A dynamic call tree τ_{dct} is a four tuple $\langle \mathcal{N}_\tau, \mathcal{E}_\tau, N_a, N_0 \rangle$ where $\mathcal{N}_\tau$ is a set of nodes, $\mathcal{E}_\tau$ is a set of edges, $N_a \in \mathcal{N}_\tau$ is the active node, and $N_0 \in \mathcal{N}_\tau$ is the root with $in(N_0) = 0$. For all nodes $N_\phi \in \mathcal{N}_\tau - \{N_0\}$, $in(N_\phi) = 1$.

The following Definition 2 defines the CCT that contains the standard components of a DCT, in addition to $\mathcal{E}_F$, the set of forward edges, $\mathcal{E}_B$, the set of back edges, and $\mathcal{N}_B$, the set of nodes that receive a back edge. We say that N_ρ *receives* a back edge when $(N_\phi, N_\rho) \in \mathcal{E}_B$. Even though τ_{cct} is not strictly a tree, we can distinguish the back edges in $\mathcal{E}_B$ from the other edges in $\mathcal{E}_\tau$. Definition 2 also states that $in(N_\phi) = 1$ for all CCT nodes $N_\phi \in \mathcal{N}_\tau - \{N_0\} - \mathcal{N}_B$ (i.e., all nodes, except for the root and those nodes that receive back edges, must have a unique parent).

Definition 2. A calling context tree τ_{cct} is a four tuple $\tau_{cct} = \langle \tau_{dct}, \mathcal{E}_F, \mathcal{E}_B, \mathcal{N}_B \rangle$, where τ_{dct} is a dynamic call tree, $\mathcal{E}_F$ is the set of forward edges, $\mathcal{E}_B$ is the set of back edges, $\mathcal{E}_\tau = \mathcal{E}_B \cup \mathcal{E}_F$ is the full set of edges, and $\mathcal{N}_B = \{N_\rho : (N_\phi, N_\rho) \in \mathcal{E}_B\}$ is the set of nodes that receive a back edge. For all nodes $N_\phi \in \mathcal{N}_\tau - \{N_0\} - \mathcal{N}_B$, $in(N_\phi) = 1$.

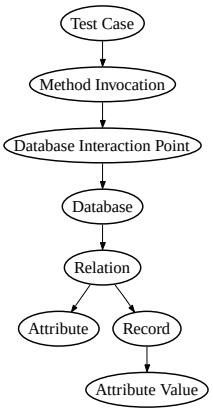

Figure 6: Database-Aware Coverage Tree.

We construct each type of coverage tree with the probes *Before*(τ, σ) and *After*(τ). The *Before* probe updates τ to include a node for a structural element σ that corresponds to a method invocation. The DCT's *Before* instrumentation (i) adds a new node to τ as a child of the active node, (ii) sets the active node to σ, and (iii) returns the updated tree to the coverage monitor. Upon completion of the method, the *After* probe for the DCT (i) updates τ by setting the new active node to the parent of the current N_a and (ii) returns the tree to the monitor.

The *Before* probe for the CCT manages the breadth of the tree by examining the children of the active node, denoted *children*(N_a), in order to determine if σ has already been called by the active node. If $N_\rho = \sigma$ for a $N_\rho \in$ *children*(N_a), then *Before* sets N_a to the node N_ρ. The CCT's *Before* probe controls the depth of τ by examining the execution context in order to find a node that is equivalent to σ. If σ is the same as a N_ϕ that is located above σ in the current tree path, then *Before* (i) designates N_ϕ as the new active node and (ii) creates a new edge from σ to N_ϕ. Refer to [2, 12] for more details about these trees and the probes.

4.4 Database-Aware Trees

The monitor can also create either a *database interaction dynamic call tree* (DI-DCT) or a *database interaction calling context tree* (DI-CCT). These database-aware trees still respectively adhere to Definitions 1 and 2. However, the nodes within a DI-DCT or DI-CCT correspond to a (i) method invocation (e.g., $\langle call\ m \rangle$), (ii) definition of a database entity (e.g., $\langle t_k[l], \textbf{define} \rangle$), or (iii) use of a database entity (e.g., $\langle rel_j, \textbf{use} \rangle$). When structural element σ corresponds to a database interaction, the coverage monitor uses four additional probes to construct the tree: *BeforeDatabaseUse*, *BeforeDatabaseDefine*, *AfterDatabaseUse*, and *AfterDatabaseDefine*. The database-aware probes can insert database entity nodes into the tree at all levels of interaction granularity. These probes subsequently invoke the appropriate *Before* and *After* based upon the type of τ (e.g., a probe calls the DCT's *Before* when updating τ_{dct}).

Figure 8 summarizes the process of database-aware coverage monitoring. After initializing the tree, the coverage monitor iteratively executes a *Before* probe, the database interaction point, and an *After* probe. These database-aware probes (i) capture the submitted SQL string, (ii) extract and analyze portions of the database's state and/or the query's result set, and (iii) update the coverage tree with nodes and edges that reflect the database interaction. If coverage is recorded at the attribute value level, then the monitoring of `FindFile`'s **select** statement in Figure 2 results in a tree containing the leaf nodes $\langle$/usr/bin/bibtex, **use**$\rangle$ and $\langle$/usr/bin/bison, **use**$\rangle$. Similarly, the **insert** statement requires the insertion of the leaves $\langle 135, \textbf{define} \rangle$ and $\langle$/usr/bin/yacc, **define**$\rangle$. Upon the completion of testing, the monitor stores the coverage tree so that it can be used to calculate test adequacy according to a variety of database-aware criteria (e.g., [12, 13, 17]).

Tree	DB?	Context	Probe Time	Tree Space
CCT	×	Partial	Low - Moderate	Low
DCT	×	Full	Low	Moderate - High
DI-CCT	✓	Partial	Moderate	Moderate
DI-DCT	✓	Full	Moderate	High

Database? $\in \{\times, \checkmark\}$
Context $\in$ {Partial, Full}
Probe Time Overhead $\in$ {Low, Moderate, High}
Tree Space Overhead $\in$ {Low, Moderate, High}

Figure 7: Comparing the Coverage Trees.

A data flow-based coverage criterion, such as *all-record-DUs*, requires the test suite to cover all of the database interaction associations like $DIA = \langle N_{def}, N_{use}, t_k \rangle$ [13]. For instance, if test case T_i causes method m to define and use the record t_k, then we can mark *DIA* as covered. Traversing a database-aware coverage tree reveals how the test cases covered the data flow-based test requirements. Intuitively, T_i covers *DIA* if the coverage tree contains the nodes $N_1 = \langle t_k, \textbf{def} \rangle$ and $N_2 = \langle t_k, \textbf{use} \rangle$ such that (i) $\langle call\ T_i \rangle$ is an ancestor of $\langle call\ m \rangle$, (ii) $\langle call\ m \rangle$ is an ancestor of nodes N_1 and N_2, and (iii) N_1 is to the left of N_2 in the tree. Our database-aware coverage tree also supports the use of McMaster and Memon's call stack coverage criterion during the process of regression testing [17]. If a test suite is being reduced or prioritized with this strategy, then each path in the coverage tree corresponds to a requirement.

4.4.1 Monitoring a Database Use

BeforeDatabaseUse employs the traditional *Before* probe to update τ and thus record that a database interaction took place at node N_r. The *AfterDatabaseUse* probe intercepts the result set $\mathcal{S}$ that consists of the attribute values that match the **select** statement.[3] Since a **select** can specify attributes from multiple relations, $\mathcal{S}$ may contain records whose attribute values are derived from one or more relations in the database. For instance, the **select** query in Figure 9 will yield a result set $\mathcal{S}$ that mixes attribute values from relations rel_j and $\widehat{rel_j}$ (i.e., we assume that A_l is an attribute of rel_j, relation $\widehat{rel_j}$ contains $\widehat{A_l}$, and Q is a predicate as defined in Figure 5).

In order to correctly preserve the database interactions, *AfterDatabaseUse* must determine the containing record and relation for each attribute value in the result set. For example, if relation rel_j contains attribute A_l and $\mathcal{S}$ includes an attribute value from A_l, then τ must have (i) nodes to represent the use of rel_j and A_l and (ii) the edge $(\langle rel_j, \textbf{use} \rangle, \langle A_l, \textbf{use} \rangle)$ to indicate that rel_j contains A_l. The *AfterDatabaseUse* probe leverages the JDBC meta-data in-

[3]This discussion focuses on the use of the database that occurs when the program submits a **select** query. The coverage monitor employs regular expressions to extract the contents of the **where** clause within the defining-using **delete** and **update** commands.

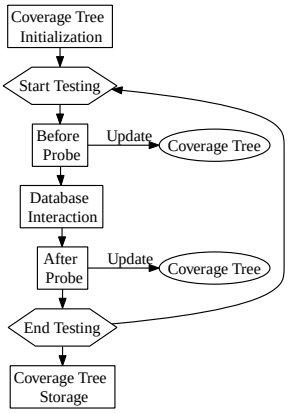

Figure 8: Database-Aware Coverage Monitoring.

terface and consults the relational schemas in order to first identify the proper content and context of each database entity and then update τ with the necessary nodes and edges.

4.4.2 Monitoring a Database Definition

After parsing the intercepted SQL statement to identify rel_j as the relation subject to definition, *BeforeDatabaseDefine* executes a **select** to extract rel_j's state. Once this probe stores rel_j, it first modifies τ so that it contains a node indicating that the database interaction took place and then it passes control back to the coverage monitor. Upon completion of the SQL command, *AfterDatabaseDefine* extracts the state of rel_j and uses differencing to determine how the program changed the relation. We perform a *symmetric relational difference* (SRD) to determine the difference(s) between relation rel_j (i.e., the relation before the interaction) and $rel_{j'}$ (i.e., the relation after the interaction). When recording coverage at the attribute value level, we use *symmetric attribute value differencing* (SAVD) to discern the difference(s) between the record t_k in rel_j and $rel_{j'}$.

The monitor uses differencing techniques because they accurately identify changes in a relation without relying upon RDBMS-specific functions such as triggers. We did not leverage the change detectors described in [4, 5] since they operate on rooted trees with node labels instead of relational data. We could not use the relational differencing system developed in [16] because the use of lossy compression may cause it to overlook a database modification and yield incorrect coverage results. Instead, the test coverage monitor uses a database-aware extension of the Myers difference algorithm [21]. Figure 10 shows the output of the SRD and SAVD techniques when an **update** statement defines the database by changing the value of $t_2[2]$ from 3 to 4. In this example, δ_{rc} denotes the set of record(s) with contents that are different in the two relations and δ_{av} stands for the set of attribute value(s) that have different values in the two input records. Even though the example in Figure 10 focuses on the **update** statement, the coverage monitor can also detect any database modifications that result from executing an **insert** or **delete** command.

4.5 Inserting the Instrumentation

Figure 11 shows a partial CFG before we introduce the instrumentation (the σ node corresponds to either a method invocation or a database interaction). If σ is a method call, then we insert a call to $Before(\tau, \sigma)$. We introduce a call

$\mathcal{S}$

select $A_l, \widehat{A_l}$
from $rel_j, \widehat{rel_j}$
where Q

A_l	$\widehat{A_l}$
1	A
3	D

Figure 9: Example of a Result Set.

rel_j

	A_1	A_2
t_1	1	2
t_2	2	3
t_3	3	4

$\delta_{rc} = \{t_2\}$
$\delta_{av} = \{t_2[2]\}$

$rel_{j'}$

	A_1	A_2
t_1	1	2
t_2	2	4
t_3	3	4

update rel_j
set $A_2 = 4$
where $A_2 = 3$

Figure 10: Differencing for the update Command.

to *BeforeDatabaseUse* or *BeforeDatabaseDefine* when σ corresponds to a database interaction. If N_r submits a SQL statement and the type is statically detectable (e.g., the **select** operation is specified in the SQL or the `executeQuery` method submits the string), then we add a call to the appropriate database-aware probe. If static analysis does not reveal whether N_r is a defining or a using SQL, then we insert a call to a *BeforeDatabase* probe that (i) dynamically determines the type of the database interaction and (ii) uses this type information to invoke either *BeforeDatabaseUse* or *BeforeDatabaseDefine*. *BeforeDatabase* invokes *BeforeDatabaseDefine* if N_r is of type defining and calls *BeforeDatabaseUse* otherwise. We also use the aforementioned technique while inserting the database-aware *After* probes.

For each method represented by CFG $G = \langle \mathcal{N}, \mathcal{E} \rangle$, we insert a call to one of the *Before* probes by first removing each edge $(N_\phi, \sigma) \in \mathcal{E}$ for all nodes $N_\phi \in pred(\sigma)$. The instrumentor also adds the edge $(N_\phi, Before(\tau, \sigma))$ to $\mathcal{E}$ for each predecessor node N_ϕ (the notation $succ(N_\phi)$ and $pred(N_\phi)$ respectively denotes the successor(s) and predecessor(s) of node N_ϕ). The insertion of the edge $(Before(\tau, \sigma), \sigma)$ completes the introduction of *Before*. We introduce an *After* probe by first removing the edge $(\sigma, N_\rho) \in \mathcal{E}$ for each node $N_\rho \in succ(\sigma)$. Next, we add the edge $(\sigma, After(\tau))$ and introduce $(After(\tau), N_\rho)$ for all nodes $N_\rho \in succ(\sigma)$.

4.6 Implementation Details

We implemented the current version of the test coverage monitor with the Java 1.5 and AspectJ 1.5 programming languages. AspectJ supports the introduction of arbitrary code segments at certain join points within a Java application. A *join point* is a well-defined location in a program's CFG. We use AspectJ to define (i) *pointcuts* that identify specific join points and (ii) bodies of *advice* that execute before and after the pointcut [15]. The test coverage monitor defines pointcuts and before and after advice that it uses to construct the DCT, CCT, DI-DCT, and DI-CCT. The coverage monitor also employs aspects that (i) initialize the coverage tree before the first test case runs and (ii) store the tree prior to the conclusion of test suite execution. Additional aspects preserve the semantics of the program under test by ensuring that all of the result sets support multiple iterations, as discussed in Section 3.2.

The static instrumentor can operate in a *batch mode* that inserts TCM probes into multiple applications during a single run. The test coverage monitor must place static instrumentation into the database-centric application each time

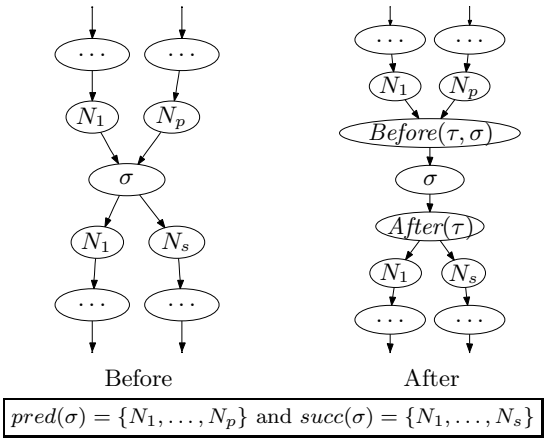

Figure 11: Inserting the Coverage Probes.

the source code of the program or the test suite changes. The dynamic approach to instrumentation introduces the probes during testing. The load-time dynamic instrumentor introduces the probes on a per-class basis with either the JVM tools interface (JVMTI) or a custom class loader. The flexibility that dynamic instrumentation affords is offset by the fact that this approach often increases the time overhead of testing more than the use of statically inserted probes.

5. EMPIRICAL EVALUATION

We designed the experiments with the intent of measuring the (i) time overhead of statically inserting the probes, (ii) impact that static instrumentation has on the space overhead of an application, and (iii) time overhead associated with test coverage monitoring. In future empirical studies, we will evaluate regression testing techniques that use the coverage results. For this experiment, we executed the static instrumentor and the coverage monitor in ten separate trials for each case study application and we report an arithmetic mean (standard deviations were uniformly small and thus we do not provide them in the data tables). We performed all of the experiments on a GNU/Linux workstation with kernel 2.6.11-1.1369, a dual core 3 GHz Pentium IV processor with 1 MB of L1 cache, and 2 GB of memory.

The empirical study used six database-centric applications that range in size from 548 to 1455 non-commented source statements (NCSS), as summarized in Figure 12 (the term "Methods" refers to any executable code body within the program, including the test cases). These case study applications employ a wide variety of strategies to test the program's interaction with a 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 that executes within the same JVM as the application itself. Since the applications use JDBC, it is possible to configure the programs to use other databases such as PostgreSQL [20] or MySQL [33]. Figure 13 shows that the test suite ranges in size from 13 to 51 test cases and the tests normally comprise between 30 and 60% of the NCSS. As indicated in Figure 14, an application contains between 7 and 45 database interaction points. Since the tests often iteratively invoke the methods that contain these interactions, the majority of the points are executed many times.

Even though our implementation of the TCM component supports dynamic instrumentation with either the JVMTI or a custom class loader, the experiments focus on measuring the performance of the class loading approach. Since most Java virtual machines (e.g., the Sun JVM and the Jikes RVM) use a similar type of class loader interface, this choice ensures that our experimental results are more likely to generalize to other execution environments. We also designed the experiment in this manner because our preliminary results revealed that the "heavy weight" JVMTI introduced significant time overheads, in confirmation of prior results [24]. We configured the static instrumentation technique to operate on the bytecode of a case study application.

Name	Classes	Methods	NCSS	Per
`Reminder (RM)`	9	55.0	548.0	*Program*
		6.11	60.89	*Class*
			9.96	*Method*
`FindFile (FF)`	5	49.0	558.0	*Program*
		9.8	111.6	*Class*
			11.39	*Method*
`Pithy (PI)`	11	73.0	579.0	*Program*
		6.64	52.64	*Class*
			7.93	*Method*
`StudentTrack (ST)`	9	72.0	620.0	*Program*
		8.0	68.89	*Class*
			8.61	*Method*
`TransactMan (TM)`	6	87.0	748.0	*Program*
		14.5	124.67	*Class*
			8.6	*Method*
`GradeBook (GB)`	10	147.0	1455.0	*Program*
		14.7	145.5	*Class*
			9.9	*Method*

Figure 12: Case Study Applications.

Application	# Tests	Test NCSS / Total NCSS
`RM`	13	227/548 = 50.5%
`FF`	16	330/558 = 59.1%
`PI`	15	203/579 = 35.1%
`ST`	25	365/620 = 58.9%
`TM`	27	355/748 = 47.5%
`GB`	51	769/1455 = 52.8%

Figure 13: Characterization of the Test Suites.

5.1 Results Analysis

Instrumentation. Figure 15 presents the time overhead associated with the static instrumentation of the case study applications. For the column labeled "All," we used batch mode to insert probes into all of the case study applications. For the smaller applications, instrumentation never takes more than 4.5 seconds. The experiments reveal that the instrumentation of a larger application (e.g., `GB`) incurs one additional second of time overhead when compared to instrumenting a smaller application (e.g., `FF`). Across all of the case study applications, static instrumentation takes 4.72 seconds on average. Batch mode completes the insertion of probes into every application in less than nine seconds. In summary, we judge that our approach to static instrumentation introduces the probes with minimal time overhead.

Opting for flexibility instead of a minimal increase in space overhead, we insert the instrumentation before and after a database interaction rather than placing probes within the JDBC driver itself (see Section 3.1 for more details about this choice). Therefore, the space overhead metric includes both (i) the bytecode instructions inserted at the boundaries of all method invocations and database interactions that exist within the program and the tests and (ii) the coverage monitoring probes themselves. Since many modern JVMs natively support Jar and Pack compressed archives [25], we

Application	executeUpdate	executeQuery	Total
RM	3	4	7
FF	3	4	7
PI	3	2	5
ST	4	3	7
TM	36	9	45
GB	11	23	34

Figure 14: Summary of Database Interactions.

App	FF	PI	RM	ST	TM	GB	All
Time (sec)	4.39	4.40	4.40	4.39	5.17	5.58	8.69

Figure 15: Static Instrumentation Time.

report the size of the bytecodes with and without the use of compression. The results in Figure 16 describe the impact of static instrumentation across all case study applications. If we consider the use of Jar, then statically inserting the probes increases space overhead by 420%. Since the instrumented application is highly compressible with Pack, it is possible to reduce the costs for the storage and network transmission of these bytecodes.

Space overhead is also high because the bytecode for a call to the *Before* probe must (i) store the contents of the program stack in local variables, (ii) check to ensure that coverage monitoring is enabled, (iii) load the probe, and (iv) invoke the probe. The instrumentor also adds additional bytecode instructions to handle any exceptions that might be thrown by a probe (e.g., we use exception handlers when the storage of a tree exceeds the available disk space). For example, the constructor for the `org.hsqldb.sample.FindFile` class contains three bytecode instructions before instrumentation and twenty-seven afterward. Finally, a noteworthy design choice of the AspectJ compiler is that it accepts an increase in static application size for a reduction in execution time by always using additional bytecode operations instead of Java's reflection mechanism [11].

Test Coverage Monitoring. During the comparison of the static and dynamic instrumentation techniques, we configured the TCM component to generate a traditional coverage tree and then we ran the test suite. Figure 17 provides the average test coverage monitoring time across all applications. The empirical results reveal that statically inserted probes increase testing time by 12.5% when the monitor creates a CCT. For the smaller applications (e.g., `RM` and `FF`), the Static-CCT configuration increases time by no more than 8%. For larger applications that contain the most method calls and database interactions (e.g., `TM` and `GB`), Static-CCT's increase is less than 15%. The results in Figure 17 also demonstrate that it is more expensive to (i) dynamically construct a tree instead of using static instrumentation and (ii) create a DCT as an alternative to a CCT. We conclude that static instrumentation enables efficient coverage monitoring when the program changes infrequently and the flexibility of dynamic monitoring comes with a moderate increase in time overhead.

For the experiments that measure how the variation of database interaction granularity impacts testing time, we always used static instrumentation to construct a database-aware CCT. As shown on the horizontal axis of the graphs in Figure 18, we executed the test coverage monitor for all applications and all levels of database interaction granularity (e.g., $\mathbf{P}$ stands for the conventional program coverage tree and $\mathbf{A_v}$ denotes a tree containing attribute values). For several applications such as `RM`, `FF`, and `GB`, we observe that

Compress	Before Instr (bytes)	After Instr (bytes)
None	29275	887609
Jar	15623	41351
Pack	5699	34497

Compression	Probe Size (bytes)
None	119205
Jar	40017
Pack	35277

Jar: $((41351 + 40017 - 15623)/15623) \times 100 = 420\%$

Figure 16: Static Space Overhead.

Instr	Tree	TCM Time (sec)	Per Incr (%)
Static	CCT	7.44	12.5
Static	DCT	8.35	26.1
Dynamic	CCT	10.17	53.0
Dynamic	DCT	11.0	66.0

Normal Average Testing Time: 6.62 sec

Figure 17: Static and Dynamic Monitoring Time.

there is only a small increase in time overhead when we vary the database interaction level (the diamonds at the top of the bars in Figure 18 indicate that the standard deviation across the trials was very small). For `RM`, `FF`, and `GB`, constructing an $\mathbf{A_v}$-level tree instead of a traditional CCT increases testing time by less than 6%. This is due to the fact that a test for these applications normally interacts with a small number of database entities.

Figures 18(c) and (d) demonstrate that monitoring time for `PI` and `ST` markedly increases as the database interaction granularity transitions from the $\mathbf{P}$ to the $\mathbf{A_v}$ level. This phenomenon occurs because both `PI` and `ST` have tests that iteratively interact with a large portion of the relational database and this type of testing behavior necessitates the insertion of many nodes and edges into the coverage tree. `PI` shows a more noticeable increase at the attribute value level than `ST` because it interacts with a relation that has three attributes while `ST`'s relation only contains two attributes. However, both `PI` and `ST` demonstrate acceptable time overheads when we record coverage at the record level (e.g., 9.317 seconds for `PI` and 9.941 for `ST`). Figure 18(e) reveals that it takes longer to record `TM`'s coverage at the $\mathbf{R_c}$-level than it does at the $\mathbf{A_v}$-level. This result is evident because `TM`'s test cases create a relation that normally contains more attributes than records.

Figure 19 provides the average value of coverage monitoring time across all applications when we varied the database interaction granularity. These results indicate that we can efficiently monitor coverage at all levels of interaction if we use static instrumentation to construct a database-aware CCT. For example, coverage monitoring at the relation level only incurs a 14.20% average increase in testing time. The TCM component can record coverage at the attribute value level with a 53.17% increase in time overhead.

Since the size of the coverage tree often increases the time and space overheads associated with monitoring, we measured tree size in terms of the (i) number of nodes and (ii) in-memory size of the tree. In order to calculate tree size, we traverse the tree in the JVM's heap and measure the size of each node using the technique described in [12, 23].

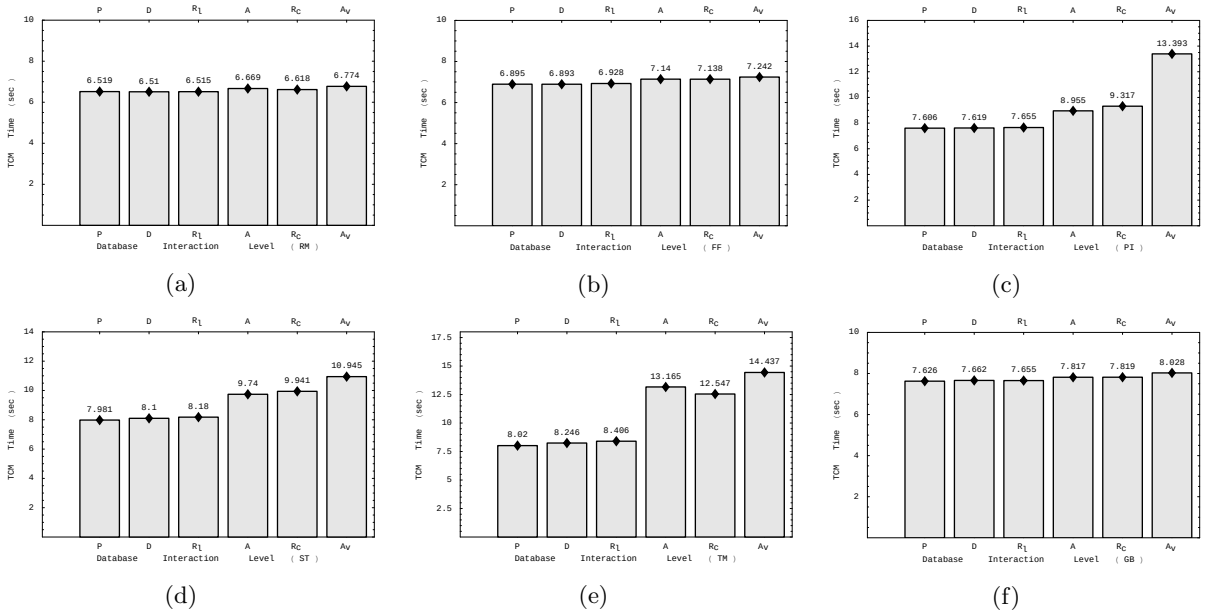

Figure 18: Test Coverage Monitoring Time at Different Database Interaction Levels.

DB Level	TCM Time (sec)	Per Incr (%)
Program	7.44	12.39
Database	7.51	13.44
Relation	7.56	14.20
Attribute	8.91	34.59
Record	8.90	34.44
Attribute Value	10.14	53.17

Figure 19: Database Interaction Granularity.

We report the tree sizes at the program, relation, record, and attribute value levels because (i) **P** provides a base line, (ii) $\mathbf{R_l}$ is structural, and (iii) $\mathbf{R_c}$ and $\mathbf{A_v}$ are state-based. The results in Figure 20 indicate that the DI-CCT is always smaller than the corresponding DI-DCT. Moreover, the $\mathbf{A_v}$-level DI-CCT is approximately the same size as the **P**-level DCT. This result suggests that the inclusion of either fine grained information about the database interactions or full execution method context will cause the same increase in the size of the coverage report. Figure 20 also shows that the $\mathbf{A_v}$-level DI-DCT contains an order of magnitude more nodes than the comparable DI-CCT. We found that only the $\mathbf{R_c}$ and $\mathbf{A_v}$-level DI-DCT introduced sufficient memory pressure to force extra invocation(s) of the JVM's garbage collector and noticeably increase coverage monitoring time.

5.2 Threats to Validity

The experiments described in this paper are subject to *validity threats*. *Internal* threats to validity are those factors that have the potential to impact the measured variables defined in Section 5.1. One internal validity threat is associated with defects in the prototype of the database-aware coverage monitor. We controlled this threat by visualizing small TCM trees and checking them to ensure correctness. We also implemented a wide variety of automated checks in order to ensure that the trees adhere to the structure described by Figure 6 and Definitions 1 and 2.

External threats to validity are factors that limit the ability to generalize the experimental results. Since the empirical studies described by this paper use a limited number of case study applications, we cannot claim that these results are guaranteed to generalize to other database-centric applications. Another threat to external validity is related to the size of the selected case study applications, test suites, and relational databases. Yet, if we compare our case study applications to those that were used in other studies by Tonella [31] (mean size of 607.5) and the Siemens application suite [26] (mean size 354.4), the average size of our programs and test suites (mean size 571) is comparable to the size of the other applications. Since our applications interact with a relational database, they may be more substantial than the programs used in these previous studies. Our study also focused on six small to moderate size applications that vary in terms of their NCSS and the type of database interaction because there are no large database-centric applications in the software-artifact infrastructure repository [8]. To further control this type of threat, we configured each application to interact with the real world HSQLDB RDBMS that is used in current applications such as OpenOffice.

6. RELATED WORK

There are many existing approaches to test coverage monitoring [14, 19, 22, 27, 30]. However, none of these techniques were explicitly designed to monitor a program's interaction with a relational database. Several recent papers have also focused on the testing and analysis of database-centric applications [6, 10, 29]. While Chays et al. present a test data generation technique that operates in a partially automated manner [6], their AGENDA framework does not address the challenges of coverage monitoring. Even though [29] presents a coverage criterion for **select** queries, they do not focus on test coverage monitoring and their approach does not handle SQL statements that define the database. The present work is also distinguished from other mechanisms for constructing call trees (e.g., [2, 17]) because our trees capture details about database coverage.

Halfond and Orso describe a test adequacy criterion that concentrates on the coverage of SQL "command forms" [10]. This paper is closely related to our work because it contains a brief description of a test coverage monitor. Whereas we

DB Level	CCT Nodes (#)	DCT Nodes (#)	CCT Size (KB)	DCT Size (KB)
Program ($\mathbf{P}$)	341	7157	125	2009
Relation ($\mathbf{R_l}$)	583	10445	179	2740
Record ($\mathbf{R_c}$)	1421	11960	818	3694
Attribute Value ($\mathbf{A_v}$)	6833	32021	2152	9108

Figure 20: Size of the Test Coverage Monitoring Tree.

designed our tree-based report to support the calculation of adequacy according to multiple criteria (e.g., [12, 13, 17]), their approach to monitoring is specifically tailored for a single adequacy criterion. Unlike our technique, the monitor in [10] does not focus on how the program and tests interact with the state and structure of the database. Moreover, Halfond and Orso do not empirically evaluate the performance trade-offs associated with their coverage monitor [10].

7. CONCLUSIONS AND FUTURE WORK

This paper presents a database-aware test coverage monitor that uses instrumentation to capture and analyze the database interactions that originate from the program and the test suite. The instrumentation probes create a coverage tree containing nodes and edges that represent a program's definition and use of relational database entities. The empirical results demonstrate that coverage monitoring increases testing time from 13% to no more than 54%, depending upon the granularity at which the monitor tracks the database interactions. As part of future work, we intend to further improve efficiency by implementing demand-driven instrumentation techniques that insert and remove probes at the method or instruction level [19]. We will also leverage the coverage results to perform database-aware regression testing tasks such as test suite reduction and prioritization [12, 17]. Finally, we will measure the performance of the coverage monitoring and regression testing tools when we use them with larger database-centric applications.

8. REFERENCES

[1] B. Alpern and et al. The Jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2), 2005.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc of PLDI*, pages 85–96, 1997.

[3] R. Bloom. Debugging JDBC with a logging driver. *Java Developer's Journal*, 2006.

[4] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proc of SIGMOD*, pages 26–37, 1997.

[5] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc of SIGMOD*, pages 493–504, 1996.

[6] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification, and Reliability*, 14(1):17–44, 2004.

[7] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc of 10th SAS*, volume 2694 of *LNCS*, pages 1–18, June 2003.

[8] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[9] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proc of 4th AOSD*, pages 51–62, 2005.

[10] W. G. J. Halfond and A. Orso. Command-form coverage for testing database applications. In *Proc of 21st ASE*, pages 69–80, 2006.

[11] E. Hillsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc of 3rd AOSD*, pages 26–35, 2004.

[12] G. M. Kapfhammer. *A Comprehensive Framework for Testing Database-Centric Applications*. PhD thesis, University of Pittsburgh, Pittsburgh, Pennsylvania, 2007.

[13] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *Proc of 9th ESEC/10th FSE*, pages 98–107, 2003.

[14] M. Kessis, Y. Ledru, and G. Vandome. Experiences in coverage testing of a Java middleware. In *Proc. of 5th SEM*, pages 39–45, 2005.

[15] G. Kiczales, E. Hillsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[16] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proc of 22nd VLDB*, pages 63–74, 1996.

[17] S. McMaster and A. M. Memon. Call stack coverage for test suite reduction. In *Proc of 21st ICSM*, pages 539–548, 2005.

[18] S. Microsystems. JDBC data access API. 2007. `http://developers.sun.com/product/jdbc/drivers/`.

[19] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proc of 27th ICSE*, pages 156–165, 2005.

[20] B. Monjian. *PostgreSQL*. Addison-Wesley, 2000.

[21] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[22] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc of the 21st ICSE*, pages 277–284, 1999.

[23] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software Practice and Experience*, 37(7):747–777, 2007.

[24] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proc of 1st AOSD*, pages 141–147, 2002.

[25] W. Pugh. Compressing Java class files. In *Proc of PLDI*, pages 247–258, 1999.

[26] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[27] V. Roubtsov. Emma: a free Java code coverage tool. `http://emma.sourceforge.net/index.html`, 2005.

[28] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Companies, Inc., New York, NY, 5th edition, 2006.

[29] M. J. Suarez-Cabal and J. Tuya. Using an SQL coverage measurement for testing database applications. In *Proc of 12th FSE*, pages 253–262, 2004.

[30] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proc of ISSTA*, pages 86–96, 2002.

[31] P. Tonella. Evolutionary testing of classes. In *Proc of ISSTA*, pages 119–128, 2004.

[32] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proc of ISSTA*, 2006.

[33] R. J. Yarger, G. Reese, and T. King. *MySQL and mSQL*. O'Reilly and Associates, Sebastopol, CA, 1999.