Dynamic Invariant Detection for Relational Databases

Jake Cobb
Georgia Institute of Technology
jake.cobb@gatech.edu

Gregory M. Kapfhammer
Allegheny College
gkapfham@allegheny.edu

James A. Jones
University of California, Irvine
jajones@ics.uci.edu

Mary Jean Harrold
Georgia Institute of Technology
harrold@cc.gatech.edu

ABSTRACT

Despite the many automated techniques that benefit from dynamic invariant detection, to date, none are able to capture and detect dynamic invariants at the interface of a program and its databases. This paper presents a dynamic invariant detection method for relational databases and for programs that use relational databases and an implementation of the approach that leverages the Daikon dynamic-invariant engine. The method defines a mapping between relational database elements and Daikon's notion of program points and variable observations, thus enabling row-level and column-level invariant detection. The paper also presents the results of two empirical evaluations on four fixed data sets and three subject programs. The first study shows that dynamically detecting and inferring invariants in a relational database is feasible and 55% of the invariants produced for each subject are meaningful. The second study reveals that all of these meaningful invariants are schema-enforceable using standards-compliant databases and many can be checked by databases with only limited schema constructs.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; H.2.1 [**Database Management**]: Schema and subschema; F.3.1 [**Logic and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs-*Invariants*

General Terms: Reliability, Verification

Keywords: Dynamic invariants, relational databases, schema modification

1. INTRODUCTION

Dynamic invariants are properties of program state that are observed to hold during one or more executions of the program. Dynamic invariants are also called *likely invariants* because they are guaranteed to hold only over this set of executions instead of over all possible executions. Dynamic invariants form an *operational abstraction* that is useful for understanding program behavior [13].[1]

Dynamic invariant detection, introduced by Ernst and colleagues and implemented as the Daikon invariant detection engine[2] [8], has been applied successfully to support a number of software-engineering tasks. Examples of such tasks include component integration testing [12], interface discovery [6], test-input generation [14], and software-behavior classification [5]. In addition to the internal in-memory state recorded by Daikon, many applications rely heavily on external state, such as configuration files and database systems. In particular, *relational database management systems* (RDBMS) based on the structured query language (SQL) standard are often used in a wide variety of applications [7, 11]. Yet, existing techniques do not compute dynamic invariants for the properties of external state.

To address this limitation of previous techniques, we developed, and present in this paper, a new approach to dynamic invariant detection for relational databases and programs that use relational databases. Our approach is based on the existing dynamic invariant detection techniques developed for the intra-program level. Although the Daikon dynamic detection approach was originally developed to compute invariants using a program's state, our implementation leverages this framework because its extensible design supports importing data from other sources [9].

Our technique captures invariants in the state of relational database tables for both application-independent, fixed data sets and dynamic data sets induced by interaction with an RDBMS during program execution. The technique is able to detect invariants for both individual columns across all rows and across multiple columns within a given row.

The main benefit of our new technique is that it collects dynamic invariants for relational databases and dependent applications, thus bringing to the database context a broad family of existing techniques for software-engineering tasks. Additionally, our method supports a new set of database-specific client applications. Section 4.2.2 presents one such application: the use of database invariants to suggest schema modifications that increase data integrity guarantees.

The main contributions of this paper are:

- A new technique that extends the notion of dynamic invariant detection to SQL-based relational databases and the applications that employ them.

[1] We use *invariant* to refer to dynamic invariants.

[2] Hereafter, we simply call this system Daikon.

WODA '11 July 18, 2011, Toronto, Canada

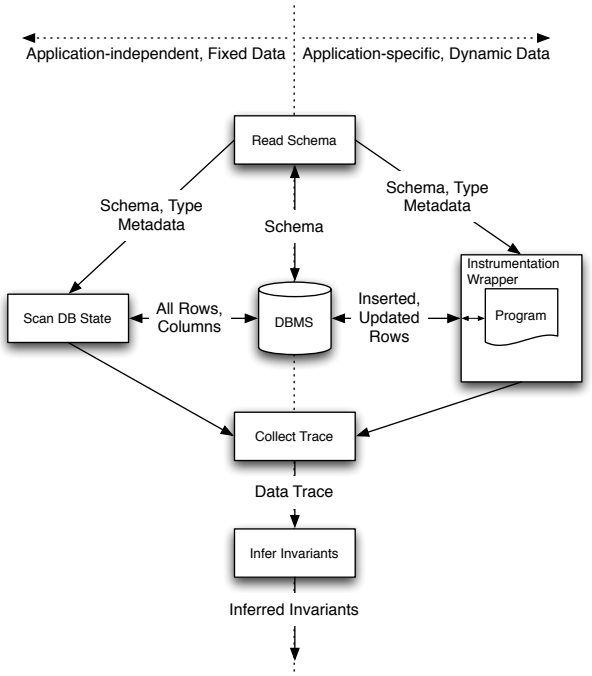

Figure 1: Process diagram depicting the database-aware approach. The left side shows the static, application-independent process, and the right side shows the dynamic, application-specific process.

- An implementation that analyzes existing databases and the Java applications that utilize these databases to automatically produces files for input to Daikon.
- Two empirical studies using seven subjects, one of which analyzes the quality of invariant detection and the other of which examines a database-specific application for schema constraint modification.

2. DATABASE-AWARE APPROACH

Program invariants are typically defined in terms of specific *program points*, which are points of interest in the program, along with the *program state*, which is usually the set of program variables that are observable at those points. For example, a developer might specify the program-points to be method entries and the program state to be parameters of the respective methods. In this case, given a method entry `void foo(int x, int y)` and a set of executions, the invariant `x < y` means that the value of `x` was always less than the value of `y` at the entry to `foo` on those executions.

To enable detection of the dynamic invariants, the program points can be *instrumented* to collect the program state. Then, the instrumented program is executed with some set of inputs (e.g., a test suite) to produce a *data trace*, which is a record of the variable values observed at program points. Potential invariants are tested against the values in the data trace. The final result is the set of invariants that were not falsified during any execution.

Our approach, which is depicted in Figure 1, offers two complementary options for collecting invariants in the context of a relational database. The first option is to infer invariants statically from the state of the database at a given point in time, independent of any program execution. The second option is to monitor the interactions between a program and its databases and infer invariants from the dynamic state of the database during program execution. The first option is useful for data sets that are not managed by an application, are shared by multiple applications, or are managed by an application that cannot be instrumented. The second option is useful when a data set is managed by a single application that is able to be instrumented.

Table 1: Mapping between database structure and program elements

Program Element	Database Element
Program Point	Table
Variable	Column
Occurrence	Row

For both options supported by our approach, Figure 1 shows that invariant detection begins by reading the database schema (*Read Schema*) to obtain structural and type information; we define and discuss the structural mapping between relational databases and program points and variables in Section 2.1. In the static case, *Scan DB State* reads the entire state of the database and produces a data trace. In the dynamic case, the *Instrumentation Wrapper* monitors communication between the database and the program and produces data traces that correspond to only new or modified rows after an initial full-state trace; we explain the representation of individual data elements in the data traces in Section 2.2. In either case, *Collect Trace* aggregates the data traces and *Infer Invariants* processes the data and yields the set of inferred invariants.

2.1 Structural Mapping

As shown in Table 1, our technique maps the database schema and structure to program concepts. The database-aware method considers each table in the database to be equivalent to an independent, non-hierarchical program point. The columns defined in the schema of each table are equivalent to named program variables. The technique treats each row of data in a database table as an occurrence or observation of variable values at a program point.

We chose to represent the relational database structure in this way to facilitate detection of two general classes of invariants. The first class consists of single-variable invariants that indicate a property that was universally true for a database column. For example, `table.a_count > 0` means that the value of the `a_count` column was always positive. The second class consists of multi-variable invariants that indicate a property that was true within each row. For example, `table.a_count < table.b_count` means the value of the `a_count` in a given row was always less than the value of `b_count` in that *same* row, not that each observed value of `a_count` was less then all observed values of `b_count`.

2.2 Data Representation

The SQL standard [10] defines a number of specific data types that are organized into broader categories such as strings and numbers. To reduce both the potential for spurious invariants and the computational cost, our technique uses type information to restrict the columns that may be tested for invariant relationships. Table 2 gives the comparable groups along with the SQL types that determine membership. In the table, the first two columns show the group number and a descriptive name, respectively. The third col-

Table 2: Data type mapping and comparability

Group	Name	SQL Types	Java Type

1	Text	`CHAR`
		`VARCHAR`
		`TEXT`
			`String`

2	Integer	`INTEGER`
		`NUMERIC`
		`BIT`[3]
			`int`

3	Decimal	`FLOAT`
		`DOUBLE`
		`REAL`
		`DECIMAL`
			`double`

4	Binary	`BLOB` `BIT`[4]	`byte[]`
5	Text Set	`SET`	`String[]`
6	Datetime	`DATETIME` `TIMESTAMP`	`String`
7	Date	`DATE`	`String`
8	Time	`TIME`	`String`
9	Interval	`INTERVAL`	`int`
10	Primary Key	`INTEGER`[5]	*reference*

umn lists the SQL types that belong to a group. The last column shows an equivalent type from the Java programming language. Comparisons for testing potential invariants can be thought of in terms of this type, with intra-group SQL type representations being converted if necessary. All columns in a given comparability group are comparable to each other, but not to members of other groups. Columns that are equivalent to an array type in Java also allow comparison of the respective elements to another group. Specifically, elements of the Binary group (4) are comparable to Integer group (2) members and elements of the Text Set group (5) are comparable to the Text group (1).

Although Table 2 shows the major SQL types for each group, the list is not exhaustive. Our technique groups SQL types that are a size variation on one of the listed types with the listed type. For example, `SMALLINT` is a size-variation of `INTEGER` and thus, it it placed in group 2. Additionally, individual relational database implementations support many non-standard types. Our technique places vendor-specific types in the group that most closely matches the data represented by that type. For example, the `ENUM` type supported by MySQL and the `CLOB` type supported by Oracle would be included in the Text group (1).

Although grouping and converting most types is straightforward, a few points are worth mentioning. Our technique represents binary values (group 4) as an array of bytes where each element is the value of a single byte in the same order as the binary stream. Although they are represented as `String`s, our technique groups dates and times separately from other textual types. Intuitively, dates and times have specific semantics that general text does not. For example, the hypothetical invariant `start_date != end_date` is potentially interesting whereas `start_date != description` is almost certainly spurious. In general, the purpose of comparable groups is to avoid detecting invariants between columns that cannot be meaningfully compared.

In the relational model, `NULL` is a special value that represents unknown or missing data. Whether or not `NULL` is a potential column value is defined by the database schema and does not depend on the data type of the column. To capture this property, our technique introduces a synthetic variable that represents the "`NULL`-ness" of the column value. The synthetic variable is equivalent to a pointer or reference type in a programming language, but it takes on only the value `null`,[6] when the corresponding column is `NULL`, or a constant value, when the column has a non-`NULL` value. Synthetic variables are introduced only for columns in which the schema permits `NULL`, thus preventing never-`null` invariants from being produced for columns that forbid `NULL` values.

There is one exception to the aforementioned type mapping rules for integer primary keys. It is common for a table's schema to use an integer as the primary key and to have the RDBMS automatically assign a unique value to it on the insertion of a new row. Thus, our technique treats columns that are integer primary keys like pointers or references instead of normal integers, and it introduces no synthetic variable even if the schema allows `NULL` values.

3. IMPLEMENTATION

Our implementation consists of two main components. The first is a Python program that collects data traces from relational databases and implements the mapping described in Section 2. The second is an extension of the P6Spy [4] database driver for Java that the technique uses to capture database invariants from Java programs.

3.1 Data Tracing

We implemented data trace collection and the mapping described in Section 2 as a Python program that produces Daikon declaration and trace files from an existing database instance. Given database connection information, it reads the schema information and constructs an internal representation of the tables and columns, including information such as the representation type, conversion routine, and synthetic `NULL` variables. When this information is first read, the data trace program writes a Daikon declarations file to disk and serializes the internal structure for reuse during tracing. The program writes Daikon declarations files in the 2.0 format.[7] It supports MySQL directly and handles several other databases through use of SQLAlchemy.

The program produces Daikon trace files by selecting all rows in the database and writing them to file according to the schema information read earlier. Because Daikon uses a text-based trace format, each representation type has a conversion function that transforms the data appropriately. For example, a `BLOB` column entry containing two 0-valued bytes must be written as `[0 0]`. By default, the trace file is filtered through GZip before being written to disk.

3.2 Database Driver Modification

P6Spy [4] is a JDBC driver for Java that wraps another JDBC driver and provides logging of SQL statements or result sets that it passes between the calling application and the driver it wraps. We wrote a new module for P6Spy and made some adjustments to its core to capture database values as they are changed by an application.

When a new connection is opened, our driver retains the connection information and invokes the data trace program

[3] When defined with a length in `int` range.

[4] When defined with a length outside of `int` range.

[5] Only when used as a `PRIMARY KEY`.

[6] `NULL` refers to the SQL type and `null` refers to the Java type.

[7] For compatibility, we implemented support for the 1.0 format as well, but did not use it for the experiments in this paper.

Table 3: Fixed data set subjects

Subject	Tables	Columns	Rows
world	3	24	5302
menagerie	2	10	19
sakila	23	131	50,086
employees	6	24	3,919,015

Table 4: Java application subjects

Subject	Tables	Columns	KLOC	Test Cases
iTrust	30	177	25.5 (Java) 8.6 (JSP)	787 787
JWhoisServer	7	57	6.7	67
JTrac	13	126	12	41

described in Section 3.1 to read the metadata and produce a declarations file and an initial data trace. If the schema information has already been gathered, the data trace program does not rewrite the declarations file or append to the trace. For all subsequent calls, the modified driver inspects the SQL statement to determine whether it may result in a new trace value. If an `INSERT` or `UPDATE` statement is executed, the driver invokes the data trace program to append trace values for the modified table before returning control to the caller. The driver does not request traces after `DELETE` or `TRUNCATE` statements because they cannot add any observed values. If the driver cannot determine whether or not a database call inserted or updated data (e.g., when a stored procedure is executed), it conservatively invokes the data trace program to append trace data for all tables.

4. EMPIRICAL STUDIES

This section presents the results of the empirical studies that we performed to evaluate our technique. First, Section 4.1 describes the empirical setup we used for the studies. Then, Sections 4.2.1 and 4.2.2 describe the two studies we performed to evaluate the invariants produced.

4.1 Empirical Setup

We evaluated our technique using seven subjects. The first four subjects are sample databases provided by MySQL.[8] We selected these subjects to evaluate whether meaningful invariants could be extracted from a database snapshot independent of a particular application's use of the database. Table 3 shows, for each of these subjects, the number of tables, columns, and rows in the second, third, and fourth columns, respectively. The `world` and `menagerie` databases contain information about international cities and a pet shop, respectively. The `employees` database contains employment data that might be used in a human resources application, and the `sakila` database contains data that models a video store chain. Both `employees` and `sakila` are used for the testing and verification of MySQL.

The last three subjects are applications that are written in Java and use a relational database. We selected these subjects to investigate whether meaningful invariants could be detected during dynamic modification of the data by a particular application. Each Java subject includes an automated test suite with test cases that interact with the respective databases. Table 4 shows, for each application, the number of tables and columns in the second and third columns, respectively, and the sizes of the code base and the test suite in the fourth and fifth columns, respectively.

[8]Available from `http://dev.mysql.com/doc/index-other.html`.

iTrust [1] is a web application that handles medical information, such as patient records.[9] iTrust uses MySQL as its RDBMS. JWhoisServer [3] is an open source implementation of the `WHOIS` protocol. JWhoisServer supports several different database systems. For this study, we configured JWhoisServer to use MySQL. JTrac [2] is an open source web application for customizable issue-tracking. JTrac supports integration with a number of external components, such as LDAP. JTrac uses the Hibernate Object-Relational Mapping framework, which provides JTrac with support for a number of database systems. For the study, we configured Hibernate to use MySQL as the back-end RDBMS.

We used the tools described in Section 3 to generate Daikon declarations and trace files. For the fixed data sets, we ran the data trace program described in Section 3.1 against the populated databases. For the Java subjects, we configured each application to use the modified P6Spy driver from Section 3.2 and then executed their respective test suites. We then ran Daikon on the resulting declarations and trace files, using the default configuration options, to produce a set of dynamic invariants for each subject. We experimented with enabling additional invariant checks and modifying parameters to the invariant checkers, but found this tends to help detection for some subjects at the expense of others. For this reason, we report only those results based on the default configuration that Daikon would use without manual tuning by the user. As mentioned in Section 5, we intend to further investigate additional invariant checking and parameter modification as part of future research.

4.2 Studies

We performed two studies to evaluate our technique. After analyzing the quality of the dynamic invariants, Section 4.2.1 illustrates the method's trade-offs by giving several concrete examples. Then, Section 4.2.2 evaluates the use of invariants for database schema constraint modification.

4.2.1 Study 1: Invariant Quality

The goal of the first study is to assess both the quantity and quality of dynamic invariant detection for relational databases. We produced invariants for each subject as described in Section 4.1. We manually inspected every invariant and identified two classes of spurious invariants, which we call vacuous and lack-of-data. *Vacuous* invariants express relationships between variables that have no semantic relationship to each other. *Lack-of-data* invariants occur because of a limited number of supplied input values; the most common form of lack-of-data invariant is variable equality with a literal value. We classified invariants as spurious by comparing each with the database schema and relevant test data set from the respective subjects.

Figure 2 shows the number of dynamically detected invariants, with the subjects listed on the vertical axis and the number of invariants shown on the horizontal axis. The bar for each subject is segmented by the number of invariants that we classified as vacuous, lack-of-data, or meaningful.

Overall, the invariant results were good across the set of subjects with the notable exception of JWhoisServer. The majority of dynamic invariants detected were not spurious. However, there is not a clear relationship between the to-

[9]iTrust was developed to support instruction in testing at North Carolina State University, and is currently maintained by the RealSearch group (`http://agile.csc.ncsu.edu/realsearch`).

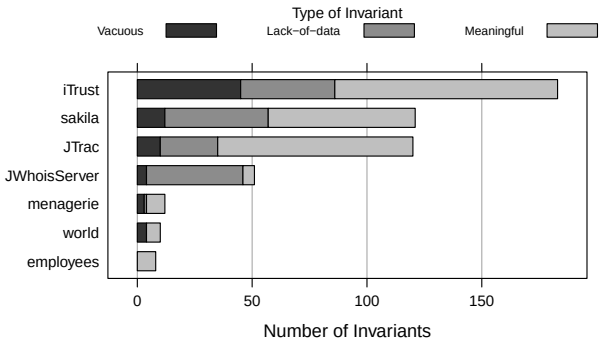

Figure 2: Dynamic invariants detected

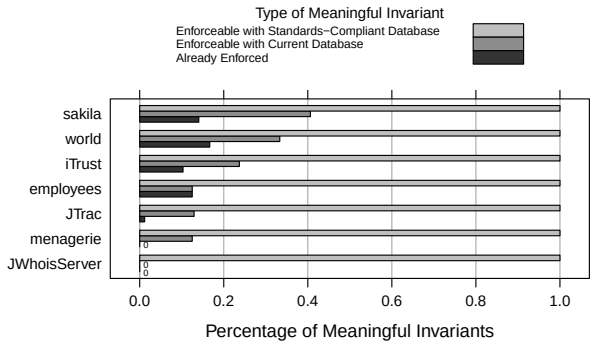

Figure 3: Schema enforcement of dynamic invariants

tal number of invariants detected and the quality of those invariants. The `employees` database had the least number of invariants, but had no spurious invariants of either type. Most of the invariants for `employees` are relationships between dates which reflect the semantic meaning of the columns. Examples of meaningful invariants include:

- `dept_emp.from_date <= dept_emp.to_date`
- `employees.gender one of { "F", "M" }`
- `employees.birth_date < employees.hire_date`

Although compared as strings, dates are formatted according to ISO 8601 so the ordering expressed in these invariants reflects a chronological relationship.

JWhoisServer had the highest percentage of spurious invariants due to lack of data because it uses a very small set of test data. Examples of lack-of-data invariants include:

- `mntnr.login == "mntnt"`
- `inetnum.changed == "2006-10-14 16:21:09"`
- `person.pcode one of { "D-12345", "NWR" }`

However, even in the case of JWhoisServer, the high number of lack-of-data invariants is useful information as it indicates the very limited data set used by the test suite.

Although it also had many meaningful invariants, iTrust fared the worst overall in terms of the percentage of vacuous invariants. iTrust also has lack-of-data invariants for the same reason as JWhoisServer in that it reuses the same test data for many different tests. Because iTrust uses database tables with a large number of comparable columns, it also ends up with many other spurious invariants. Examples of vacuous invariants include:

- `patients.lastName >= patients.address1`
- `patients.phone1 <= patients.BloodType`
- `cptcodes.Description != cptcodes.Attribute`

There is clearly no semantic relationship between a patient's phone number and blood type. However, these columns were compared because both were defined as textual, resulting in the meaningless, although true, lexicographical order expressed in this dynamic invariant.

The excellent quality of the `employees` invariants hints at two important aspects of invariant detection: large enough samples and a wide variety of data. Each of the Java subjects re-used small sets of test data, which was reflected in the absence of spurious invariants.

4.2.2 Study 2: Schema Enforcement

The goal of this study is to evaluate the effectiveness of applying the invariants for data integrity through database schema modification. The schema of a relational database specifies constraints on the values of its constituent columns, both through the SQL type and by auxiliary constraints such as `NOT NULL`. There may also be implicit constraints on the data that are assumed or enforced by the database user. Making these implicit constraints explicit in the schema provides a much stronger assurance of data integrity as the RDBMS will enforce the constraints and reject data that violates a schema constraint.

Using the same subjects and invariants as our first study, we manually inspected each invariant and identified those that could possibly be enforced by the relational database's schema. For each potentially enforceable invariant, we compared it with the schema definition to see if it could be enforced by modifying the schema or it reflected a constraint that was already explicit in the schema.

Figure 3 shows the proportion of invariants in each category, with the subjects listed on the vertical axis and the number of invariants shown on the horizontal axis. For each subject, the black, bottommost bar shows the percentage of invariants that were already enforced by the schema and the dark-gray, middle bar shows the percentage of invariants where we recommend a schema modification to enforce the invariant. We took into account the results of Section 4.2.1 when deciding whether a schema modification was appropriate. For example, we did not include a (`NULL`-able) column that was never `NULL` when the table to which it belongs suffered lack-of-data invariants. For this reason, the percentages shown in Figure 3 are relative to the total number of invariants remaining after removing the spurious invariants from the first study. Additionally, we only suggest schema modification for constraints that are supported by the specific RDBMS used by the respective subjects. For example, virtually all of the invariants could be enforced by a `CHECK` constraint, but MySQL does not support the `CHECK` constraint so we do not count these invariants in the Enforceable bar for subjects that used MySQL. The light-gray, topmost bar for each subject in Figure 3 reflects the fact that all of these invariants could be enforced by an RDBMS that complies with the SQL standard [10].

Common examples of schema-enforced invariants include positive integers where the type was unsigned and fixed sets of string values resulting from `ENUM`s. Table 5 contains several examples of invariants that were already enforced by the database schema. In future work, it would be straightforward to filter out invariants already in the schema.

Table 6 shows examples of schema modifications that enforce invariants. Adding a `NOT NULL` constraint to a column

Table 5: Example invariants enforced by the schema

Subject	Invariant	Schema Definition
employees	employees.gender one of { "F", "M" }	ENUM('F','M')
world	countrylanguage.IsOfficial one of { "F", "T" }	ENUM('F','T')
sakila	customer.active one of { 0, 1 }	TINYINT(1)
sakila	inventory.film_id >= 1	SMALLINT(5) UNSIGNED
JTrac	spaces.guest_allowed one of { 0, 1 }	BIT(1)

Table 6: Example schema modifications for invariant enforcement

Subject	Invariant	Original Schema	Modified Schema
iTrust	isnull(message.message) != null	TEXT	TEXT NOT NULL
sakila	isnull(film_text.description) != null	TEXT	TEXT NOT NULL
JTrac	isnull(history.time_stamp) != null	DATETIME	DATETIME NOT NULL
JTrac	user_space_roles.user_id >= 1	BIGINT(20)	BIGINT(20) UNSIGNED
menagerie	pet.sex one of { "f", "m" }	CHAR(1)	ENUM('M','F')
world	country.Population >= 0	INT(11)	INT(11) UNSIGNED
employees	isnull(titles.to_date) != null	DATE	DATE NOT NULL

definition is one of the most common possible schema modifications. Because synthetic variables are introduced to represent NULL values only for columns without the NOT NULL constraint, invariants of the form isnull(column) != null[10] are never the result of already existing schema enforcement.

5. CONCLUSION AND FUTURE WORK

This paper extends the notion of dynamic invariant detection from software programs to relational databases and programs that use relational databases. We defined a mapping between relational database structure and data types to enable dynamic invariant detection with the Daikon engine. We also conducted two empirical studies and provided analyses that demonstrate the feasibility of collecting meaningful invariants for four fixed data sets and three subject programs. Finally, we showed that dynamic invariants for relational databases can be effectively used to identify areas where the database schema constraints could be tightened to protect data integrity.

Whereas this paper provides a good baseline for integration of dynamic invariant detection with relational database models, there are a number of areas for future work. First, the mapping of the database structure could be extended to allow for additional invariant detection on relationships between columns in different tables or for the results of JOINs executed by a particular application. Second, additional information could be used to filter or enhance the invariants that we are testing. For instance, schema or WHERE-clause enforced invariants could be removed from consideration, allowing the rest of the schema-enforceable invariants to be presented to the user as suggestions for schema modification.

Third, the studies presented in this paper do not address the performance of the implementation. As such, we intend to create efficient techniques for capturing values as they flow through the database driver, rather than consulting the RDBMS after a modification. Moreover, because our preliminary experiments suggest that the tuning of the parameters of the invariant checkers had an unpredictable impact on the subjects, we will focus on this issue in a more comprehensive empirical study. Finally, the various applications of dynamic invariant detection in software engineering, such as component integration testing and test-case generation, could be applied in the relational database context. Combining the technique presented in this paper with our future work will result in a complete and efficient engine for dynamic invariant detection with both databases and the programs that interact with databases.

[10]The synthetic variables are shown this way for clarity; they are named differently in the implementation

6. REFERENCES

[1] iTrust. http://agile.csc.ncsu.edu/iTrust.

[2] JTrac. http://www.jtrac.info.

[3] JWhoisServer. http://jwhoisserver.net/.

[4] P6Spy. http://www.p6spy.com/.

[5] J. F. Bowring, M. J. Harrold, and J. M. Rehg. Improving the classification of software behaviors using ensembles of control-flow and data-flow classifiers. Technical Report GIT-CERCS-05-10.

[6] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface specifications. In *Proceedings of the 28th ICSE*, 2006.

[7] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, fifth edition, 2007.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Transactions on Software Engineering*, 27(2), 2001.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[10] International Organization for Standardization. *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. 1992.

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

[12] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of COTS-component-based software. In *Proceedings of the 29th ICSE*, 2007.

[13] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th ESEC and the 11th FSE*, 2003.

[14] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th ECOOP*, 2005.