Model Checking Database Applications

Milos Gligoric[1] and Rupak Majumdar[2]

[1] University of Illinois at Urbana-Champaign, USA
[2] Max Planck Institute for Software Systems, Germany
gliga@illinois.edu, rupak@mpi-sws.or

Abstract. We describe the design of DPF, an explicit-state model checker for database-backed web applications. DPF interposes between the program and the database layer, and precisely tracks the effects of queries made to the database. We experimentally explore several implementation choices for the model checker: stateful vs. stateless search, state storage and backtracking strategies, and dynamic partial-order reduction. In particular, we define independence relations at different granularity levels of the database (at the database, relation, record, attribute, or cell level), and show the effectiveness of dynamic partial-order reduction based on these relations.

We apply DPF to look for atomicity violations in web applications. Web applications maintain shared state in databases, and typically there are relatively few database accesses for each request. This implies concurrent interactions are limited to relatively few and well-defined points, enabling our model checker to scale. We explore the performance implications of various design choices and demonstrate the effectiveness of DPF on a set of Java benchmarks. Our model checker was able to find new concurrency bugs in two open-source web applications, including in a standard example distributed with the Spring framework.

1 Introduction

We present the design, implementation, and evaluation of DPF, an explicit-state model checker for database-backed web applications. Most web applications are organized in a three-tier architecture consisting of a presentation tier, a business-logic tier, and a persistent database tier. In a typical usage scenario, a user of the application starts a session, makes one or more requests to the application, and then closes the session. The processing of a request depends on the state of the database and can modify the database. The application server hosting the web application assigns a new thread for each request, and runs the logic implemented in the business tier to handle the request. The thread may access the data tier to store or retrieve information to/from a database. After each request is handled, the response is sent back to the user. Since the underlying http protocol is stateless, the application threads typically do not share the state directly. Instead, all state is stored in the session object and in the persistent store. In particular, requests by different users (or requests made in different sessions) only share the database and no other shared state. Most modern languages provide frameworks that simplify the development of web applications (e.g., Spring and Grails for Java, Django for Python, Rails for Ruby).

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 549–564, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Since web applications concurrently process requests made by multiple users, there is the potential for concurrency bugs. One class of bugs are atomicity violations, where the application may perform several sequential interactions with the database, without ensuring the sequence occurs atomically. This can expose inconsistent states to the database. Consider two users getting a record and then concurrently deleting it; the second attempt to delete may fail. Note that such bugs can arise even when the implementation of the database management system (DBMS) correctly implements ACID semantics for each transaction. Existing techniques for checking race conditions and atomicity violations [7, 26] in multi-threaded code cannot be applied directly, as they usually depend on explicit tracking of synchronization operations or on heap cells that are read or written.

Since bugs in web applications can have high financial costs, it is reasonable to consider developing systematic state-space exploration tools that look for property violations. Moreover, the application domain makes model checking [2, 12, 14] especially attractive: each request handler typically makes few interactions with the database, to improve latency of the application. Thus, techniques such as partial-order reduction are expected to work exceptionally well: each thread can atomically run all its code between two database transactions. At the same time, developing such a model checker presents new and non-trivial technical challenges: How can we represent the state of the database in the model checker? How can we store and restore states, particularly in the presence of a large amount of data in the database? How do we perform partial-order reduction while respecting the database semantics?

We focus on model checking the business and data tiers of web applications, assuming the correctness of the database management system. We have implemented a model checker DPF (for *Database PathFinder*) for Java programs.

A straightforward model checking approach would model the database interactions using reads and writes on shared-memory objects representing the objects implemented in the database and use standard explicit-state model checking [12, 14, 28]. Unfortunately, we show that such an approach does not scale. Database queries have complex semantics, and their correct modeling brings in too many details of the database implementation. Instead, we represent the application as a multi-threaded imperative program interacting with a database through a core SQL-like declarative query language, and precisely model the semantics of a relational database in the semantics of the programming language. That is, the model checker represents database state as a set of relations, and directly models integrity constraints on the data, such as primary key constraints. The actual database is run along with the model checker to store the concrete relations.

We explore several design choices in our model checker. First, we explore stateful vs. stateless search. In stateful search, we implement two approaches for backtracking the database state. In the first approach, we exploit the savepoint and rollback mechanism of the database, so we can roll back the database state at a backtrack point. In the second approach, we replay the queries performed on the database from an initial state to come to a backtrack point.

Second, we explore partial-order reduction strategies at various granularity levels. Partial-order reduction (POR) requires identifying when two operations are dependent. We give conditions to identify dependent operations at the database, relation, attribute,

record, and cell levels. Dependencies are more precise as we go down the levels from database to cell, but require more bookkeeping. We experimentally evaluate the effect of POR at different levels: we show that the number of states explored decreases from naïve exhaustive exploration to cell level, with over $20\times$ reduction in some cases.

Our implementation and evaluation on 12 programs suggests that model checking can be an effective tool for ensuring correctness of web applications. In our experiments, we were able to find concurrency errors in two open-source Java-based web applications. Specifically, we found concurrency errors in PetClinic, an example e-commerce application distributed with the Spring framework, and in OpenMRS, an open-source medical records system. In each case, the programmers had not considered conflicting but non-atomic accesses made concurrently to the database.

While much of our model checker specializes general model-checking techniques, our contributions include: (1) design choices that customize the model checker (including stateful/stateless search, state storage, and backtracking) to the domain of database-backed applications (Section 3), (2) domain-specific versions of partial-order reduction (Section 4), and (3) empirical validation that model checking can be quite effective in detecting concurrency errors for database-backed applications (Section 5).

2 Modeling Database Applications as Transition Systems

We formalize our model-checking algorithm for a concurrent imperative language that accesses a relational database using a simplified structured query language (SQL). We model the semantics of the database as in [5], but additionally allow multiple threads of execution in the program interacting with the database.

Relational Databases. A *relational schema* is a finite set of relation symbols with associated arities. Each relation symbol is an ordered list of named *attributes*; an attribute is used to identify each position. A record is an ordered list of attribute values. The value of attribute A has position $\mathsf{pos}(A)$. A finite relation is a finite set of records. For simplicity of exposition, we assume that each attribute value is an integer; our implementation handles all the datatypes supported by a database.

A *relational database* over a relational schema S represents a mapping from relation symbol $\mathbf{r} \in \mathsf{S}$ to finite relation r, such that r has the same arity as $\mathbf{r}$. We write $r \cup \{\rho\}$ to denote an extension of the relation r with the record ρ, which has the same arity as r, and $r \setminus \{\rho\}$ to denote a removal of the record ρ from the relation r. A *key* can be defined on a relation symbol $\mathbf{r}$ to identify an attribute that has a unique value in each record of the relation; $\mathsf{kdef}(\mathbf{r})$ holds if a key is defined on $\mathbf{r}$ and $\mathsf{key}(\mathbf{r})$ returns the position of that key. Finally, for a relational schema S, a relation symbol $\mathbf{r} \in \mathsf{S}$, a finite relation r of the same arity as $\mathbf{r}$, and a database R over S, we write $R[\mathbf{r} \leftarrow r]$ for the relational database where $\mathbf{r}$ is mapped to r, while all other relation symbols are the same as in R.

Structured Query Language (SQL). We assume that the program communicates with the database using a declarative query language. We focus on a simplified data manipulation language that allows querying, insertion, deletion, or update of the relations in the database. The syntax of the language is as follows.

1. The SELECT statement SELECT $A_1,\ldots,A_k$ FROM $\mathbf{r}$ WHERE ψ queries the database and returns a relation with attributes $A_1,\ldots,A_k$, such that for each record r in the relation, there is a record in the relation $\mathbf{r}$ that agrees with r on $A_1,\ldots,A_k$ and also satisfies the predicate ψ.
2. The INSERT statement INSERT INTO $\mathbf{r}$ VALUES $(v_1,\ldots,v_n)$ inserts a new record in the relation $\mathbf{r}$, if the integrity constraints are satisfied.
3. The DELETE statement DELETE FROM $\mathbf{r}$ WHERE ψ removes all records from the relation $\mathbf{r}$ that satisfy the predicate ψ.
4. The UPDATE statement UPDATE $\mathbf{r}$ SET $A = F(A)$ WHERE ψ updates the values of an attribute by applying the function F in all records that satisfy the predicate ψ, if the integrity constraints are satisfied.

Note that insertions and updates check integrity constraints on the database. These are invariants on the database that are specified at the schema level. For example, the PRIMARY KEY constraint requires that each value of a key attribute appears at most once in a relation.

We formalize the semantics of the SQL statements in a straightforward way and additionally include support for integrity constraints. We fix a schema S consisting of a set of relation symbols. A database R over S consists of a set of relations, one for each relation symbol $\mathbf{r}$ in S of the same arity as $\mathbf{r}$ and with the same attributes.

We first define some standard functions. For a relation $r = R(\mathbf{r})$ and predicate ψ over the attributes of $\mathbf{r}$, we define the *selection function* $\sigma_\psi(r)$ as a relation that includes all records that satisfy the condition ψ: $\{\rho \mid \rho \in r \wedge \rho \models \psi\}$. The *projection* $\pi_{A_1,\ldots,A_k}(r)$ projects a relation r to only the attributes $A_1,\ldots,A_k$. The substitution $r[A \leftarrow F(A)]$, for a function F mapping integers to integers is defined as $\{\langle v_1,\ldots,v_{i-1},F(v_i),v_{i+1},\ldots,v_n\rangle \mid \langle v_1,\ldots,v_n\rangle \in r \wedge i = \mathsf{pos}(A)\}$.

Lastly, for an attribute K, we define a predicate $\xi_K(r)$ that holds iff each record of r has a unique value on attribute K, i.e., for all $\rho \in r$, $\langle \rho_{\mathsf{pos}(K)}\rangle \notin \pi_K(r \setminus \rho)$.

The semantics of each SQL statement can now be given as a transformer on the database and an output relation (representing the result of the SQL statement).

Select: For a given set of attributes $\{A_i \mid i \in \{1,\ldots,k\}\}$ of a relation $\mathbf{r}$, the select operation returns a pair of the unmodified relational database and a set of records that satisfy the predicate ψ and projected on $A_1,\ldots,A_k$: $\langle R,\pi_{A_1,\ldots,A_k}(\sigma_\psi(R(\mathbf{r})))\rangle$.

Insert: For a record $\langle v_1,\ldots,v_n\rangle$ of the same arity as $\mathbf{r}$, the insert operation returns a relational database that includes a new record in the mapping for relation $\mathbf{r}$ if a key is not defined on $\mathbf{r}$, or the record has a unique value on the key attribute; otherwise, it returns the original relational database and an empty output relation:

$$\begin{array}{l}\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \cup \{\langle v_1,\ldots,v_n\rangle\}],\emptyset\rangle, \text{ if } \neg\mathsf{kdef}(\mathbf{r}) \ \vee \ \xi_{\mathsf{key}(\mathbf{r})}(R'(\mathbf{r})) \\ \langle R,\emptyset\rangle \text{ , otherwise.}\end{array}$$

Delete: The delete operation creates a new mapping for the relation symbol $\mathbf{r}$, which includes all the records that do not satisfy the predicate ψ and an empty output relation: $\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \setminus \sigma_\psi(R(\mathbf{r}))],\emptyset\rangle$

Update: The update operation creates a new mapping for the relation symbol $\mathbf{r}$, which includes all records that do not satisfy the predicate ψ and all records that satisfy the predicate ψ with substituted value for A. The update is possible if either a key is not

defined on $\mathbf{r}$ or the integrity constraints are satisfied after the update; otherwise, the operation does not modify the relational database:

$$\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \setminus \sigma_\psi(R(\mathbf{r})) \cup \sigma_\psi(R(\mathbf{r}))[A \leftarrow F(A)]], \emptyset\rangle, \text{ if } \neg\mathsf{kdef}(\mathbf{r}) \ \vee \ \xi_{\mathsf{key}(\mathbf{r})}(R'(\mathbf{r}))$$
$$\langle R, \emptyset\rangle \text{ , otherwise.}$$

Multithreaded Database Programs. Our program model consists of multithreaded imperative programs with a fixed number of threads, where each thread has its own local variables (including relation-valued variables). In addition to usual assignment, conditional, and looping constructs, each thread can make SQL statements to a shared database with a fixed schema S. We assume w.l.o.g. that there is no global shared memory state (but our implementation supports additional global state as well). In order to model transactions, we assume that a statement can be enclosed within a keyword "transaction," and the database implementation guarantees that the entire transaction executes atomically. We omit a concrete syntax for our programs.

The state of a program consists of the state of the relational database and local states of each thread:

State of the program = State of the database × States of the threads
States of the threads = a map from each thread to its local state
Local state = Location in the program × a map from each local var to its value

At the beginning of the execution of a program, each local state is in an initial state, where the program counter is at the starting location for each thread and each value is initialized to the default value of the appropriate type. The initial state of the database is provided by the user and represents the state at the beginning of an execution. Note that the result of the exploration depends crucially on the initial database state.

The execution of a program advances by performing the following steps in a loop: (1) select a thread non-deterministically from the set of live threads, and (2) execute the statements of the thread until the thread finishes the execution or is about to exit from a transaction statement (this represents a commit of the transaction); the sequence of statements executed by a thread without interruption is called a *transition* [8]. The program execution ends when all threads finish the execution.

Note that while two transactions can be interleaved in a concrete run of a program, we rely on the correctness properties of the database implementation to run transactions atomically and in isolation. Moreover, we rely on a lower-level primitive that performs retries in case a transaction must be aborted, so that we only check execution paths of the program in which transactions commit.

Explicit-State Model Checking. Now that we have modeled a database application as a state-transition system, we can implement an explicit-state model checker that systematically explores all interleavings of the program [12, 14, 28]. Since the database is the only shared state, the only *visible* operations of the program (i.e., points at which thread switches must be scheduled) are transaction statements.

In the next two sections, we discuss key implementation choices in the model checker: (1) representation of the database (in-memory vs. on-disk), (2) state storage, checkpointing and restoration, and matching, and (3) partial-order reduction.

3 Design Decisions and Implementation

Extending JPF. We implemented our model checker DPF (Database PathFinder) for Java programs as an extension of Java PathFinder (JPF) [16, 28]. JPF is a popular, extensible and configurable model checker for Java programs; a user can select stateful or stateless search, state storing and restoring mechanism, state matching algorithm, search heuristic, etc. JPF implements a backtrackable Java Virtual Machine that explores a program by interpreting bytecode instructions. As a consequence, JPF does not support (i.e., cannot analyze) Java programs that employ *native* methods, since the native methods are not implemented as java bytecode. To overcome the limitations one has to implement native methods to operate on JPF's internal memory representation.

Although in principle JPF should be able to explore any Java program, including database applications, there are significant challenges in applying it in our context. First, JPF is unable to execute database applications because applications use native methods, e.g., from Java Database Connectivity (JDBC) used by Java programs to interact with databases. Second, JPF sees accesses to shared-memory objects as the only source of non-determinism.[1] Therefore, JPF does not consider database transactions as the scheduling points, which are the key scheduling points for database applications. Thus, JPF will not explore their interleavings. Third, JPF may perform incorrect execution of database applications. JPF stores the in-memory state of an application at each scheduling point and restores the state when it explores the next choice from that point. This ignores the state of the database (and of external files). Similarly, JPF stores hashes only of in-memory objects, and will ignore the database state when matching the state in stateful exploration. Next, we describe how we customized JPF to address these challenges and enable the (optimized) exploration of database applications.

Design 1: In-Memory Database. The simplest approach to address all the challenges is to configure a database application to use an in-memory database, e.g., H2[2], which uses data structures in memory to represent a database but exposes a SQL interface to these data structures. Consequently: 1) there are no accesses to external resources (although we still had to implement a few native methods used by H2), 2) the database becomes, from the JPF perspective, an in-memory object shared among threads and therefore JPF schedules all relevant threads at each access to the data structures that represent the database, 3) the database is stored/restored by JPF at scheduling points as any other in-memory object, and 4) state matching hashes the state of the database (at the *concrete* level of the data structures that implement the database rather than at the *abstract* level of the database).

However, this approach has a number of drawbacks. JPF explores the implementation of the database together with an application, therefore introducing unnecessary overhead. We would rather explore the semantics of the application assuming correctness of the DBMS implementation. Next, keeping a database in JPF memory is not acceptable for any realistic application with many records. Also, state matching is not optimal, since many internal structures are part of the state (e.g., different orders of two records lead to different states, even when they encode the same relation).

[1] JPF also supports data non-determinism, but that is irrelevant for our discussion.

[2] `http://h2database.com/`

Our experimental evaluation showed that this technique does not complete in reasonable time or space for any real application, and not even for micro-benchmarks.

Design 2: On-Disk Database. To enable JPF to work with on-disk databases, we first intercept the native methods from the JDBC API and extended JPF to view some of these methods as scheduling points. We next extend JPF to restore the database, and so support stateless exploration even when the application updates the database. Our first approach to tackle this problem was to intercept all method invocations that access the database and save the SQL operations used in these accesses. We keep a mapping from each memory state that JPF encounters during exploration to the sequence of SQL operations executed up to that state. When JPF restores the memory state, we additionally first restore the database to what it was at the beginning of the exploration and then replay the saved SQL operations that correspond to the state being restored. While this approach correctly restores the database for each state, we recognized that in some cases we may further optimize state restoring by leveraging the roll back mechanism of databases. In our second approach, for each new state in JPF, we set a *savepoint* [3]. Then, when JPF restores the memory state, we instruct the database to roll back to the appropriate savepoint. Note that the second approach works only for depth-first search exploration because there can be at most one sequence of savepoints in a database.

To further optimize the exploration, we consider stateful search. Before each state matching, we compute the hash of the database and add it to the hash that JPF computes for the memory state. We explore two ways of hashing databases: the *full* approach that computes hash of the entire database each time, and the *incremental* approach that updates the hash value each time the database is changed. Note that the incremental hash function must be commutative [15, 20]; otherwise, the same set of records may lead to different hash values if they are inserted in different order. Our current implementation modifies the H2 database to support incremental hashing.

4 Partial-Order Reduction

POR [8, 11, 12, 30] is an optimization technique that exploits the fact that many paths are redundant as they execute independent transitions in different orders. Two transitions are *independent* [11] if their executions do not affect each other, i.e., if the two transitions commute and do not disable each other. The *naïve* approach (i.e., without POR) trivially considers any two operations to be dependent and exhaustively explores the entire transition graph. POR techniques identify dependent transitions and explore a set of paths that is a subset of the paths that are executed by the naïve approach.

For database applications, we can track dependencies among SQL operations at different granularity levels: *database*, *relation*, *attribute*, *record*, and *cell*. These levels differ in precision of the tracked information (thus enabling more pruning in the exploration) and in the cost of tracking that information (the more precise ones are more expensive to track). We now describe the dependency conditions for these levels.

Figure 1 compactly presents the sufficient conditions to identify dependent transitions for more precise granularities. These conditions assume that there is one SQL operation per transition and that both transitions use the same relation symbol $\mathbf{r}$ (as

Transitions	Granularity			
	Relation	Attribute	Record	Cell
$\langle \mathsf{S}^1,\mathsf{S}^2 \rangle$	$false$	$false$	$false$	Attribute $\wedge$ Record
$\langle \mathsf{S}^1,\mathsf{I}^2 \rangle$	$true$	$true$	$\sigma_{\psi^1}(\{\langle v_1^2,\ldots,v_n^2 \rangle\}) \neq \emptyset$	
$\langle \mathsf{S}^1,\mathsf{D}^2 \rangle$	$true$	$true$	$\sigma_{\psi^1}(R(\mathbf{r})) \cap \sigma_{\psi^2}(R(\mathbf{r})) \neq \emptyset$	
$\langle \mathsf{S}^1,\mathsf{U}^2 \rangle$	$true$	$A^2 \in \{A_1^1,\ldots,A_k^1\}$	$\sigma_{\psi^1}(R(\mathbf{r})) \neq \sigma_{\psi^1}(R^{\mathsf{U}^2}(\mathbf{r}))$	
$\langle \mathsf{I}^1,\mathsf{I}^2 \rangle$	$true$	$true$	$false \curlyvee \pi_K(\{\langle v_1^1,\ldots,v_n^1 \rangle\}) \cap \pi_K(\{\langle v_1^2,\ldots,v_n^2 \rangle\}) \neq \emptyset$	
$\langle \mathsf{I}^1,\mathsf{D}^2 \rangle$	$true$	$true$	$\sigma_{\psi^2}(\{\langle v_1^1,\ldots,v_n^1 \rangle\}) \neq \emptyset$	
$\langle \mathsf{I}^1,\mathsf{U}^2 \rangle$	$true$	$true$	$\sigma_{\psi^2}(\{\langle v_1^1,\ldots,v_n^1 \rangle\}) \neq \emptyset \curlyvee \pi_K(\{\langle v_1^1,\ldots,v_n^1 \rangle\}) \cap \pi_K(R^{\mathsf{U}^2}(\mathbf{r})) \neq \emptyset$	
$\langle \mathsf{D}^1,\mathsf{D}^2 \rangle$	$false$	$false$	$false$	
$\langle \mathsf{D}^1,\mathsf{U}^2 \rangle$	$true$	$true$	$\sigma_{\psi^1}(R(\mathbf{r})) \neq \sigma_{\psi^1}(R^{\mathsf{U}^2}(\mathbf{r})) \curlyvee \sigma_{\psi^2}(R(\mathbf{r})) \neq \sigma_{\psi^2}(R^{\mathsf{D}^1}(\mathbf{r}))$	
$\langle \mathsf{U}^1,\mathsf{U}^2 \rangle$	$true$	$A^1 = A^2$	$\sigma_{\psi^1}(R(\mathbf{r})) \neq \sigma_{\psi^1}(R^{\mathsf{U}^2}(\mathbf{r})) \vee \sigma_{\psi^2}(R(\mathbf{r})) \neq \sigma_{\psi^2}(R^{\mathsf{U}^1}(\mathbf{r})) \curlyvee$ $(\pi_K(R^{\mathsf{U}^1}(\mathbf{r})) \setminus \pi_K(R(\mathbf{r}))) \cap (\pi_K(R^{\mathsf{U}^2}(\mathbf{r})) \setminus \pi_K(R(\mathbf{r}))) \neq \emptyset$	

Fig. 1. Conditions to identify dependent transitions where $\mathbf{r}^1 = \mathbf{r}^2$

transitions on different relations are trivially independent in our language). Recall that $\mathsf{kdef}(\mathbf{r})$ holds if the relation symbol has a key; if so, we assume the key is called K.

Figure 1 uses the following notation. A pair of transitions is named by the first letter of their SQL operations, e.g., $\langle \mathsf{S}^1,\mathsf{I}^2 \rangle$ stands for $\langle \mathsf{SELECT}^1,\mathsf{INSERT}^2 \rangle$. Recall that these operations have parameters; we use X^1 and X^2 to refer to the parameters of the operations, e.g., in an expression $\sigma_{\psi^1}(\{\langle v_1^2,\ldots,v_n^2 \rangle\})$, ψ^1 is the predicate used in the first transition and $v_1^2,\ldots,v_n^2$ are the attributes used in the second transition. $R^{\mathcal{O}}$ refers to the database after the operation $\mathcal{O}$ is executed. Finally, symbol $\curlyvee$ splits a condition in the part sufficient if a primary key is not defined on the relation and the part that is additionally sufficient if a primary key is defined; other than that, $\curlyvee$ is equivalent to logical $\vee$ operator; the conditions are weaker for relations that have keys because the implicit constraints preclude some insert/update operations.

Relation Granularity. Most pairs of transitions are (conservatively) marked as dependent. For example, consider $\langle \mathsf{S}^1,\mathsf{I}^2 \rangle$ (the second row in Figure 1), which are dependent if the record to be inserted by I^2 would be selected by S^1. Because the only available information at the relation granularity level is the name of the relation used in the operations, it is not possible to know if S^1 would select the record inserted by I^2. However, the relation granularity is still more precise than the naïve exploration for the two cases ($\langle \mathsf{S}^1,\mathsf{S}^2 \rangle$ and $\langle \mathsf{D}^1,\mathsf{D}^2 \rangle$) when the transitions are always independent. First, a SELECT operation does not modify the state of the database, so two SELECT operations are independent (this is similar to read-read independence in shared-memory programs). Second, two DELETE operations are independent, because a DELETE operation either removes all the records that satisfy the predicate or does not affect the database if no record satisfies the predicate (Section 2). Thus, two DELETE operations commute, and the set of removed records is the union of the sets of records removed by these operations. Note that we do not consider all options of databases (e.g., foreign keys, observing failing operations, etc.), which may change the notion of dependence in some cases. For example, two DELETE statements may not commute if a foreign key is defined because the first delete may remove the records such that the second delete cannot execute. Also,

while database and relation granularities are the same for our language when $\mathbf{r}^1 = \mathbf{r}^2$, these granularities may differ when other options of databases are considered.

Attribute Granularity. The attribute granularity is more precise than the relation granularity, i.e., the transitions that are dependent according to the attribute granularity are dependent according to the relation granularity, while the opposite may not hold. At the attribute granularity level, the extracted information includes a set of attributes used by each transition. Consider two rows in Figure 1 ($\langle \mathsf{S}^1, \mathsf{U}^2 \rangle$ and $\langle \mathsf{U}^1, \mathsf{U}^2 \rangle$) when the attribute granularity may give more precise result. S and U are dependent if the attribute whose values are to be updated is in the set of attributes to be selected. Similarly, U and U are dependent if both update the same attribute; note that even when the attribute is the same, the transitions may actually be independent if they modify different records.

Record Granularity. The record granularity is never less precise than the relation granularity but is incomparable to the attribute granularity. The conditions that are sufficient for two transitions to be dependent are as follows. For $\langle \mathsf{S}^1, \mathsf{I}^2 \rangle$, it suffices to check if the record to be inserted by I^2 would be selected by S^1. $\langle \mathsf{S}^1, \mathsf{D}^2 \rangle$ requires that at least one of the records to be deleted by D^2 be in the set of records to be selected by S^1. $\langle \mathsf{S}^1, \mathsf{U}^2 \rangle$ requires that the sets of records selected by S^1 before and after the update differ. If a key is not defined, $\langle \mathsf{I}^1, \mathsf{I}^2 \rangle$ are never dependent because both records can be inserted in the database. However, if a key is defined, these transitions are dependent if the values of the keys to be inserted are the same. $\langle \mathsf{I}^1, \mathsf{D}^2 \rangle$ requires that the record to be inserted by I^1 would be deleted by D^2. If a key is not defined, $\langle \mathsf{I}^1, \mathsf{U}^2 \rangle$ are dependent if the record to be inserted would be updated; if a key is defined, the transitions are dependent if the record to be inserted and any updated record have the same key value. $\langle \mathsf{D}^1, \mathsf{U}^2 \rangle$ requires different set of records to be selected by D^1 before and after the update if a key is not defined. If a key is defined, the transitions are dependent if a set of records to be selected by U^2 before and after delete is different. For $\langle \mathsf{U}^1, \mathsf{U}^2 \rangle$, if a key is not defined, different sets of records should be selected using the condition of one operation on the database before and after the other operation is executed. If a key is defined, the transitions are dependent if they would insert any records that have the same key value.

Cell Granularity. The cell granularity combines the power of attribute and record granularities to identify values in the records that are accessed by each operation. This makes the cell granularity the most precise. The conditions are conjunction of conditions required for attribute and record granularities.

We implemented dependency analysis for relation and cell granularities in DPF. Since the set of relation symbols used in the SQL statements does not depend on the database content, our implementation caches the set of relation symbols for each operation and uses the cache in dependency analysis for the relation granularity.

POR vs. Database Management System (DBMS) Serializability. DBMS checks if two transitions are serializable when they execute concurrently [3]. In contrast, DPF checks if two transactions *commute* even if they are executed serially. DBMS employs mechanisms, such as read/write sets [3], to answer the serializability question. While it may be possible to use these mechanisms to check dependence, our straightforward implementation of dependence-tracking using DBMS conflict detection was imprecise

as the read/write sets involved implementation-specific objects such as locks that may not affect dependence. Thus, we implemented the semantic notions described above.

5 Experimental Evaluation

We now present a performance evaluation of DPF and describe three bugs we found.

Configurations. First, we evaluate DPF with in-memory database. Second, we evaluate DPF with on-disk database combined with one of two approaches to restore the database (Replay or Rollback), one of two approaches to hash the database (Full or Incremental), and one of three POR granularity levels (Naïve, Relation, or Cell). Therefore, we have 13 configurations of DPF. All experiments were performed on a machine with a 4-core Intel Core i7 2.70GHz processor and 4GB of main memory, running Linux version 3.2.0, and Java Oracle 64-Bit Server VM, version 1.6.0_33.

Benchmarks. Our set of benchmarks includes four real-world applications and 7 kernels that we used to evaluate DPF. The applications are as follows: `OpenMRS`[3] is an open-source enterprise electronic medical record system platform with 150K lines of Java code in the core repository; `PetClinic`[4] is an official sample distributed with the Spring framework [27] and implements an information system to be used by a veterinary clinic to manage information about veterinarians, pet owners, and pets; `RiskIt`[5] is an insurance quote application with 13 relations, 57 attributes, and more than 1.2 million records; and `UCOM`[6] is a program for obtaining statistics about usage of a system. `RiskIt` and `UCOM` have been used in previous research studies on database applications [13,23,24]. The kernels include these: `InsertDelete` is created to test the $\langle$I,D$\rangle$ dependency; `IndAtts` is created to test POR with the attribute granularity by spawning a couple of threads that update values of different attributes; `IndCells` is created to test POR with the cell granularity by spawning multiple threads that use different cells; `IndD` is created to test the dependency among delete operations; `IndRels` is created to test POR with the relation granularity; `Accesses` spawns threads that perform many (independent and dependent) SQL operations; and `Entries` is created to test our two approaches for hashing the database.

Tests. As for other dynamic techniques, DPF requires an input that initiates the exploration. While our kernels do not require any input, we had to construct test inputs for four applications. Unfortunately, none of the applications include concurrent test cases. (A concurrent test case spawns two or more threads.) However, all applications include a (large) number of sequential test cases. We created concurrent test cases by combining the existing sequential test cases; each sequential test case is executed by one thread. Combining sequential test cases can be challenging and currently we mostly do it manually. In the future, we would like to investigate in more detail the power of concurrent test cases that are obtained by combining sequential test cases. Also, we would like

[3] `http://openmrs.org` (version 1.9.1)
[4] `http://static.springsource.org/docs/petclinic.html` (revision 616)
[5] `https://riskitinsurance.svn.sourceforge.net` (revision 96)
[6] `http://sourceforge.net/projects/redactapps` (version of October 14, 2012)

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.
Benchmark ▼	DB Type ▶	In-memory	On-disk											
	DB Hashing ▶	na	Full						Incremental					
	DB Restore ▶	na	Replay			Rollback			Replay			Rollback		
	Granularity ▶	na	Naïve	Rel.	Cell	Naïve	Rel.	Cell	Naïve	Rel.	Cell	Naïve	Rel.	Cell
	Statistics ▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼	▼
Applications														
OpenMRS	Expl [ms]	-	53513	3201	3082	145881	3121	3137	53482	2634	3095	149579	3098	3109
	Speedup [X]	na	na	16.72	17.36	na	46.74	46.50	na	20.30	17.28	na	48.28	48.11
	#States	-	33855	2193	1625	33855	2193	1625	33855	2193	1625	33855	2193	1625
	Reduction [X]	na	na	15.44	20.83	na	15.44	20.83	na	15.44	20.83	na	15.44	20.83
	#Trans	-	68182	3221	2214	68182	3221	2214	68182	3221	2214	68182	3221	2214
	Memory [MB]	-	437	102	104	460	106	104	437	102	102	442	102	104
	Hash [ms]	na	89	15	9	100	10	16	17	2	4	27	1	1
PetClinic1	Expl [ms]	-	107225	20334	14223	94448	18188	13343	106107	20976	14682	93750	18714	14056
	Speedup [X]	na	na	5.27	7.54	na	5.19	7.08	na	5.06	7.23	na	5.01	6.67
	#States	-	39571	10919	5646	39571	10919	5646	39571	10919	5646	39571	10919	5646
	Reduction [X]	na	na	3.62	7.01	na	3.62	7.01	na	3.62	7.01	na	3.62	7.01
	#Trans	-	52733	13962	6708	52733	13962	6708	52733	13962	6708	52733	13962	6708
	Memory [MB]	-	610	503	387	541	515	393	617	499	393	577	509	390
	Hash [ms]	na	295	73	56	305	84	57	27	16	37	31	11	36
PetClinic2	Expl [ms]	-	4549	3359	3589	3709	2785	3005	4647	3495	3657	3599	2822	2976
	Speedup [X]	na	na	1.35	1.27	na	1.33	1.23	na	1.33	1.27	na	1.28	1.21
	#States	-	1234	458	451	1234	458	451	1234	458	451	1234	458	451
	Reduction [X]	na	na	2.69	2.74	na	2.69	2.74	na	2.69	2.74	na	2.69	2.74
	#Trans	-	1744	620	609	1744	620	609	1744	620	609	1744	620	609
	Memory [MB]	-	167	102	102	102	102	105	101	102	106	106	102	102
	Hash [ms]	na	19	12	10	11	16	9	4	2	2	2	0	3
RiskIt	Expl [ms]	-	1012343	615514	391515	1050192	632422	397537	26447	25969	25805	26509	25982	26635
	Speedup [X]	na	na	1.64	2.59	na	1.66	2.64	na	1.02	1.02	na	1.02	1.00
	#States	-	706	276	203	706	276	203	706	276	203	706	276	203
	Reduction [X]	na	na	2.56	3.48	na	2.56	3.48	na	2.56	3.48	na	2.56	3.48
	#Trans	-	1349	415	278	1349	415	278	1349	415	278	1349	415	278
	Memory [MB]	-	368	368	367	367	368	366	368	368	368	368	366	368
	Hash [ms]	na	167227	95498	58708	168815	100173	62133	7	2	5	9	5	7
UCOM	Expl [ms]	-	173510	39289	39262	177845	41901	41631	12102	9439	9545	11999	9426	9883
	Speedup [X]	na	na	4.42	4.42	na	4.24	4.27	na	1.28	1.27	na	1.27	1.21
	#States	-	1152	433	416	1152	433	416	1152	433	416	1152	433	416
	Reduction [X]	na	na	2.66	2.77	na	2.66	2.77	na	2.66	2.77	na	2.66	2.77
	#Trans	-	3421	822	775	3421	822	775	3421	822	775	3421	822	775
	Memory [MB]	-	371	370	370	370	370	370	371	365	371	371	365	371
	Hash [ms]	na	35141	6664	6705	36121	7320	7153	8	0	2	14	2	5
Kernels														
InsertDelete	Expl [ms]	4427	1078	1072	1075	1085	1078	1102	1097	1079	1080	1050	1061	1081
	#States	898	40	36	36	40	36	36	40	36	36	40	36	36
	#Trans	1416	66	54	54	66	54	54	66	54	54	66	54	54
	Memory [MB]	177	72	72	72	72	72	72	57	72	72	57	57	57
	Hash [ms]	na	1	0	2	1	3	4	0	0	1	0	2	0
IndAtts	Expl [ms]	723195	4742	4344	5219	5482	5273	5821	4665	4755	5320	5523	5215	6003
	#States	168561	3269	3153	3057	3269	3153	3057	3269	3153	3057	3269	3153	3057
	#Trans	361251	9975	8561	8369	9975	8561	8369	9975	8561	8369	9975	8561	8369
	Memory [MB]	428	165	284	141	165	236	165	165	155	141	236	198	229
	Hash [ms]	na	50	57	72	43	58	71	8	15	28	10	23	27
IndCells	Expl [ms]	115328	2314	1811	2512	2368	2244	2497	2325	2292	2558	2391	2218	2562
	#States	26693	1061	981	921	1061	981	921	1061	981	921	1061	981	921
	#Trans	64472	3055	2405	2285	3055	2405	2285	3055	2405	2285	3055	2405	2285
	Memory [MB]	249	102	102	105	102	99	102	104	99	102	105	99	104
	Hash [ms]	na	11	10	17	12	6	7	1	1	11	3	2	21
IndD	Expl [ms]	-	19133	12693	13174	45553	16009	17970	18835	12414	13117	44644	16051	17381
	#States	-	15610	11732	11732	15610	11732	11732	15610	11732	11732	15610	11732	11732
	#Trans	-	58277	35005	35005	58277	35005	35005	58277	35005	35005	58277	35005	35005
	Memory [MB]	-	294	352	372	284	369	390	202	285	284	284	327	298
	Hash [ms]	na	38	42	34	44	46	51	17	17	26	22	34	26
IndRels	Expl [ms]	113317	1902	2326	2499	2452	2287	2382	2375	2219	2418	2438	2176	2332
	#States	26693	1061	921	921	1061	921	921	1061	921	921	1061	921	921
	#Trans	64472	3055	2285	2285	3055	2285	2285	3055	2285	2285	3055	2285	2285
	Memory [MB]	254	102	102	102	102	105	104	102	104	104	104	105	105
	Hash [ms]	na	7	13	6	14	15	7	0	3	9	3	5	8
Accesses	Expl [ms]	-	18396	10231	3170	14363	8689	3069	18365	10860	3219	14376	8347	3084
	#States	-	14856	8759	369	14856	8759	369	14856	8759	369	14856	8759	369
	#Trans	-	26125	11884	590	26125	11884	590	26125	11884	590	26125	11884	590
	Memory [MB]	-	154	327	84	214	286	72	194	381	72	139	223	65
	Hash [ms]	na	93	68	26	102	57	19	28	20	29	18	18	16
Entries	Expl [ms]	-	4675	3751	3190	3442	2900	3014	4215	3218	2979	2856	2495	2628
	#States	-	467	302	139	467	302	139	467	302	139	467	302	139
	#Trans	-	819	426	215	819	426	215	819	426	215	819	426	215
	Memory [MB]	-	102	105	102	155	155	105	105	72	74	104	72	72
	Hash [ms]	na	400	324	223	405	314	252	2	2	5	4	3	6

Fig. 2. Exploration statistics for multiple DPF configurations

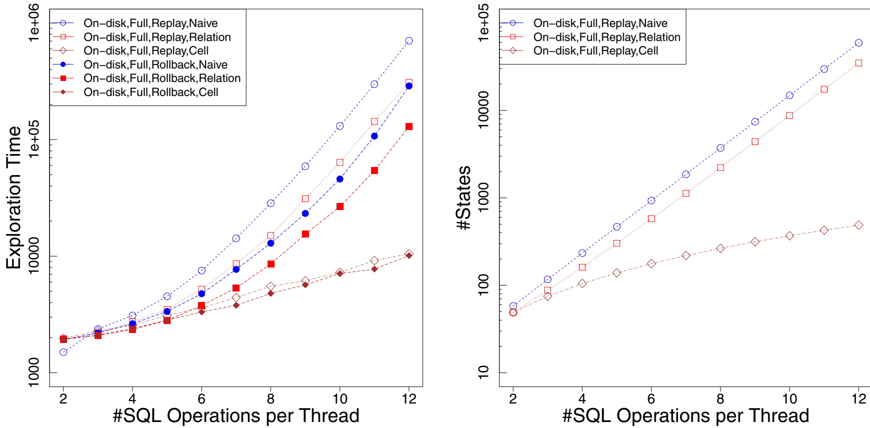

Fig. 3. Exploration time (left) and number of states (right) for the `Entries` benchmark

to combine DPF with the existing approaches [6, 18] to discover entry points in web applications and run multiple threads that use these points.

Results. Figure 2 shows, for each benchmark (listed in column 1) several statistics (column 2) for each of the 13 evaluated configurations of DPF (columns 3–15). Specifically, we show the exploration time, speedup in time (over Naïve approach), number of explored states, reduction in the state space (over Naïve approach), number of transitions, memory usage, and time for hashing the database; because of space limit we show speedup and reduction only for the applications.

We can observe the following. (1) In-memory database is not acceptable even for small examples, e.g., comparing columns 3 and 4 for `IndCells`, the exploration time is over 50x slower for in-memory database. In fact, using in-memory database, DPF often runs out of memory or time limit (set to 1h), marked with "-". (2) More precise POR granularity can significantly reduce the exploration time, e.g., looking at columns 4 to 6 for `Accesses`, going from Naïve to Relation reduced the time 2x and going from Relation to Cell reduces the time 3x more. However, more precise POR granularity does not always result in smaller exploration time because more precise POR granularity has additional cost to compute the dependency more precisely, e.g., for `IndCells` columns 5 and 6, the time for Relation is smaller than the time for Cell. Therefore, less precise POR could perform better when the number of explored states is small and there are no independent accesses, which is almost never the case for real applications [12]. Additionally, if there is a task that should be performed at each state, more precise granularity yields significantly better results even for small number of states, e.g., for `RiskIt`, columns 4 and 6 show significant improvement compared to columns 13 to 15. (3) Using Rollback to restore the database is not always faster than Replay, e.g., for `OpenMRS` columns 4 and 7, the time for Replay is smaller than the time for Rollback. We noticed that Rollback does well if the state space graph is closer to being a tree (i.e., the ratio of number of states and transitions is closer to 1). (4) Incremental always takes less time than Full for the hashing process itself, and when hashing time becomes a substantial

part of exploration, Incremental also significantly reduces the exploration time, e.g., compare `RiskIt` columns 4 and 10.

Scalability. Figure 3 illustrates scalability of the selected DPF configurations. We show the plots only for `Entries`, because of the space limit, where we parametrized the code to have an increasing number of SQL operations per thread. The plots depict the exploration time (left) and the number of explored states (right) for different number of SQL operations in each thread. (Note that the y axis is in logarithmic scale.) It can be seen that more precise granularity level scales better.

Bugs. While performing the experiments, DPF discovered one bug in `OpenMRS` and two bugs in `PetClinic`. We reported two bugs to the developers [4]. The bugs manifest as uncaught exceptions with a specific schedule of database transactions. The exception in `OpenMRS` happens if two users access the same concept class (e.g., Test, Drug, etc.) and one of the users edits and saves (or deletes) the concept after the second user has already deleted the concept. A bug in `PetClinic` is similar: the exception happens when two users attempt to delete the same pet of the same owner simultaneously. The following schedule leads to the bug: both users access the same owner then the same pet of the owner, and then one sends delete request after delete request by another user has been completed. The second user to send the delete request will get an exception.

We have found a second bug in `PetClinic` when it is configured to use database access through Hibernate. An exception happens if two users access the same owner then the same pet (the execution can be the same as in the bug that we have already reported), and then send delete requests simultaneously. In the delete handler the object is first taken from the database (SQL select) and then deleted (SQL delete), however these two operations are non-atomic and if both users first get the object and then try to delete only the first delete succeeds while the second throws an exception. This bug differs from the first schedule described above. In the first case, the bug occurs when two delete requests on the same object are performed sequentially. In the second, the bug occurs when there is a context switch point after the select of one request when the second request runs to completion, and then the first request fails to delete.

6 Related Work

Detecting Concurrency Issues Related to External Resources. Paleari et al. [22] proposed a dynamic approach to detect dataraces in web applications that interact with databases. The approach analyzes a log file of a single run and identifies dependencies among SQL queries based on the set of relations and attributes that are read/written. The solution ignores program semantics, thus leading to false alarms. Our dependency analysis is more precise and it is used to optimize model checking of database applications without false alarms. Closely related work by Zheng and Zhang [31] applies static analysis to detect atomicity violations in external resources, such as files and databases, in application servers. The difference between their work and ours is the usual distinction between static program analysis and model checking: static analysis can be less precise (i.e., have false positives); but model checking requires setting up the environment to uncover bugs. Since web application code often contains complex language features

such as reflection, building up of queries using string operations, etc., static analysis is likely to be imprecise in this domain.

Test Generation for Web Applications. Symbolic techniques have been used in generating test cases for database-backed applications [5, 17, 24] and in detecting vulnerabilities [6, 29]. In contrast to these papers, which focus on data non-determinism for a single thread of the application server, we focus on concurrent interactions of multiple server threads with the database. We assume correctness of DBMS implementation; orthogonal research looks for bugs in databases and transaction models [9, 19].

Model-Checking Tools. Explicit-state software model checking [2, 12, 14, 21, 28] has been shown useful for finding concurrency bugs. Our contribution is to apply the techniques to the important domain of web applications, and to adapt shared-memory model-checking techniques to checking database interactions. QED [18] is a model checker for web applications that systematically explores sequences of requests to a web application and looks for taint-based vulnerabilities but does not interleave transactions within a request. Artzi et al. [1] described an explicit-state model checker for web applications that does not consider concurrent requests. Petrov et al. [25] developed a tool for detecting data races in client-side web applications.

7 Conclusions and Future Work

DPF is the first step toward scalable systematic exploration tools for database-backed web applications, and much work remains. DPF can be extended to support: (1) other database constraints, e.g., foreign key, that can semantically affect even relations that do not syntactically appear in a SQL operation, (2) operations with multiple relations, e.g., join clause, (3) transactions with multiple SQL operations, and (4) exploration of transactions that can be aborted. In addition, dynamic exploration of closed programs in DPF can be combined with static techniques [31], data non-determinism [5, 24], and automatic generation of environment models [10].

Acknowledgments. We thank Don Batory, Yegor Derevenets, Sarfraz Khurshid, Hank Korth, Darko Marinov, Roland Meyer, Francesco Sorrentino, and Samira Tasharofi for discussions about this work; Mark Grechanik, Kai Pan, Kunal Taneja, and Tao Xie for information about benchmarks; and Aleksandar Milicevic for comments on the text. This material is based upon work partially supported by the US NSF under Grant Nos. CCF-1012759 and CCF-0746856. The first author was an intern at the Max Planck Institute for Software Systems during Summer 2012.

References

1. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Softw. Eng. 36(4), 474–494 (2010)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008)
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)

4. DPF home page, http://mir.cs.illinois.edu/~gliga/projects/dpf/
5. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: ISSTA, pp. 151–162 (2007)
6. Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G.: Toward automated detection of logic vulnerabilities in web applications. In: USENIX Security Symposium, pp. 143–160 (2010)
7. Flanagan, C., Freund, S.: Atomizer: A dynamic atomicity checker for multithreaded programs. Sci. Comput. Program. 71(2), 89–109 (2008)
8. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121 (2005)
9. Fonseca, P., Li, C., Rodrigues, R.: Finding complex concurrency bugs in large multi-threaded applications. In: EuroSys, pp. 215–228 (2011)
10. Giannakopoulou, D., Păsăreanu, C.S.: Interface Generation and Compositional Verification in JavaPathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
11. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
12. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL, pp. 174–186 (1997)
13. Grechanik, M., Csallner, C., Fu, C., Xie, Q.: Is data privacy always good for software testing? In: ISSRE, pp. 368–377 (2010)
14. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
15. Iosif, R.: Exploiting heap symmetries in explicit-state model checking of software. In: ASE, pp. 254–261 (2001)
16. JPF home page, http://babelfish.arc.nasa.gov/trac/jpf/
17. Khalek, S.A., Khurshid, S.: Systematic testing of database engines using a relational constraint solver. In: ICST, pp. 50–59 (2011)
18. Martin, M., Lam, M.S.: Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In: USENIX Security Symposium, pp. 31–43 (2008)
19. Mentis, A., Katsaros, P.: Model checking and code generation for transaction processing software. Concurr. Comput.: Pract. Exper. 24(7), 711–722 (2012)
20. Musuvathi, M., Dill, D.L.: An Incremental Heap Canonicalization Algorithm. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 28–42. Springer, Heidelberg (2005)
21. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
22. Paleari, R., Marrone, D., Bruschi, D., Monga, M.: On Race Vulnerabilities in Web Applications. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 126–142. Springer, Heidelberg (2008)
23. Pan, K., Wu, X., Xie, T.: Database state generation via dynamic symbolic execution for coverage criteria. In: DBTest, pp. 1–6 (2011)
24. Pan, K., Wu, X., Xie, T.: Generating program inputs for database application testing. In: ASE, pp. 73–82 (2011)
25. Petrov, B., Vechev, M., Sridharan, M., Dolby, J.: Race detection for web applications. In: PLDI, pp. 251–262 (2012)
26. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
27. Spring Framework home page, http://springsource.org/
28. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)

29. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: ISSTA, pp. 249–260 (2008)
30. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient Stateful Dynamic Partial Order Reduction. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 288–305. Springer, Heidelberg (2008)
31. Zheng, Y., Zhang, X.: Static detection of resource contention problems in server-side scripts. In: ICSE, pp. 584–594 (2012)