The Journal of Systems and Software 84 (2011) 2253–2264

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Checking enforcement of integrity constraints in database applications based on code patterns

Hongyu Zhang[a,b,*], Hee Beng Kuan Tan[c], Lu Zhang[d], Xi Lin[a,b], Xiaoyin Wang[d], Chun Zhang[d], Hong Mei[d]

[a] Key Laboratory for Information System Security (Tsinghua University), Ministry of Education, Beijing 100084, China
[b] Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China
[c] Nanyang Technological University, Singapore 639798, Singapore
[d] Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China

ARTICLE INFO

Article history:
Received 5 July 2010
Received in revised form 4 June 2011
Accepted 16 June 2011
Available online 5 July 2011

Keywords:
Integrity constraint enforcement
Code patterns
PHP
Static analysis
Code quality

ABSTRACT

Integrity constraints (including key, referential and domain constraints) are unique features of database applications. Integrity constraints are crucial for ensuring accuracy and consistency of data in a database. It is important to perform integrity constraint enforcement (ICE) at the application level to reduce the risk of database corruption. We have conducted an empirical analysis of open-source PHP database applications and found that ICE does not receive enough attention in real-world programming practice. We propose an approach for automatic detection of ICE violations at the application level based on identification of code patterns. We define four patterns that characterize the structures of code implementing integrity constraint enforcement. Violations of these patterns indicate the missing of integrity constraint enforcement. Our work contributes to quality improvement of database applications. Our work also demonstrates that it is feasible to effectively identify bugs or problematic code by mining code patterns in a specific domain/application area.

1. Introduction

Database applications constitute one of the largest software application domains. A database application is a data-intensive software system that supports business processes or functions through maintaining a database using a database management system (DBMS). In database application development, integrity constraints are used to ensure integrity of a database (Ramakrishnan and Gehrke, 2000; Silberschatz et al., 2005). These constraints include key constraints, referential constraints and domain constraints. A key constraint specifies a key that uniquely identifies a record in a database table. A referential constraint (also called a foreign key constraint) ensures that two database tables maintain a primary-key-to-foreign-key relationship. A domain constraint defines the possible range of values of an attribute. Breaking these constraints may lead to the loss of data dependencies, runtime exceptions, security flaws or even corruption of the database. Therefore, it is vital to enforce integrity constraints for database applications.

There are two levels of integrity constraint enforcement (ICE): the DBMS level and the application level. Traditionally, most DBMS systems can only enforce a limited range of constraints. Referential constraint enforcement is often not supported due to performance concerns. In recent years, some DBMSs (such as MySQL+InnoDB) can be configured to enable full support of ICE. However, even though DBMSs support ICE, many developers fail to use it. Blaha (2001) studied about 50 database applications and found that 90% of them fail to use referential constraints.

The other level of ICE is at the application level, where ICE is implemented by program code before performing any database operation that may lead to a violation. At the application level, ICE is maintained by programmers and independent of the DBMS choice. Therefore, even if the DBMS does not support ICE, the data integrity can still be kept. The risk of database corruption is thus reduced. We believe that a good database application should be "defensive" to prevent violations from happening. The lack of the application level ICE in a database application often indicates bad programming practice or even bugs. In this paper, we propose a pattern based approach for checking enforcement of integrity constraints at the application level.

In recent years, many approaches have been proposed to detect bugs or problematic code based on code patterns (e.g., (FindBugs, 2011; Hovemeyer and Pugh, 2004; PMD, 2011)). We adopt a similar approach, but focus on only the domain of database applications. We find that in database applications, code that implements ICE generally exhibits certain empirical patterns. We have identified four such patterns, namely the key constraint enforcement pattern, the referential constraint enforcement pattern for insertion

* Corresponding author. Tel.: +86 10 62773275.
E-mail address: hongyu@tsinghua.edu.cn (H. Zhang).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.06.044

and update, the referential constraint enforcement pattern for deletion, and the input validation pattern. By detecting violations of these code patterns through static analysis, we can check whether constraint enforcement has been performed in a database application. We can then achieve a better understanding of the quality of the database application and reduce the risk of database corruption. Even if ICE is implemented at the DBMS level, knowing the absences of ICE at the application level could help software maintainers identify potential ICE problems during software evolution, where the database schema or even the DBMS may change.

We have conducted experiments on nine real-world open source PHP database applications. We find that, in reality, ICE does not receive enough attention in programming practice. Implementations of ICE are often neglected at both the DBMS level and the application level. We applied our code pattern based ICE detection approach to these applications and discovered many violations of integrity constraints with high accuracy.

We believe that our approach could help improve the quality of database applications. The organization of the rest of this paper is as follows. In Section 2, we introduce integrity constraint enforcement together with an illustrative example. In Section 3, we describe four ICE-related code patterns, which can be used to detect violations of ICE at the application level. We present our experiments on nine PHP database applications in Section 4. Section 5 discusses the related work and Section 6 concludes this paper.

2. Background

2.1. Integrity constraint enforcement (ICE)

A unique feature of database applications is the enforcement of integrity constraints. Integrity constraints are conditions specified on a database schema (Ramakrishnan and Gehrke, 2000; Silberschatz et al., 2005). They restrict the data that can be stored in an instance of the database. Key constraints, referential constraints and domain constraints are three major forms of integrity constraints.

A key constraint specifies a unique key (an attribute or a set of attributes) that uniquely identifies each record in a database table. A Primary Key (PK) is a unique key whose value cannot be null. According to key constraints, no two distinct records in a table can have the same value for the PK attribute.

A referential constraint (also called foreign key constraint) identifies an attribute (or a set of attributes) in one (referencing) table that refers to an attribute (or a set of attributes) in another (referenced) table. It establishes a relationship between the two tables. For an attribute defined as a Foreign Key (FK), it should refer to a PK in the referenced table. Thus, a row in the referencing table cannot contain key values that do not exist in the referenced table.

In a database, a domain constraint is a basic form of integrity constraint that defines the possible range of values of an attribute. It includes rules for the allowed data type and length, the NULL value acceptance, etc. All of these rules are used to ensure the validity and consistency of data.

Integrity constraints should be enforced when the database is to be modified. For key constraints, whenever a record containing a PK value is to be inserted or updated, a check for key duplication is required. If the table already contains a record having the same PK value, the key constraint is violated and the operation should be rejected. To enforce referential constraints, whenever a record containing a FK value is to be inserted or updated, we need to check whether the referenced table contains the associated PK value. Whenever a record containing a PK value is to be deleted, a typical action is to delete any associated record containing a FK with the same PK value.

Domain constraints contain domain-specific rules for preserving data validity. It is not easy to use one specific approach to check all forms of the rules in a domain. However, since ultimately the data to be stored in the database comes from the external environment such as user input, we can implement input validation to help ensure domain constraints. Input validation enforces that the input submitted from the external environment must satisfy the required domain constraints before it is accepted to the database. Input validations are especially important for Web-based database applications, where security vulnerabilities such as SQL injection are real concerns.

ICE can be performed by a DBMS such as SQL Server 2005 or MySQL+InnoDB. For example, the deletion of a PK record may cause the deletion of the corresponding FK records. Changes to the data that violates the constraints can be automatically rejected. However, if the constraints are not explicitly specified in the database schema or the DBMS is not configured to enable automatic checking of constraints, the integrity of the database would be in danger. Empirical studies (Blaha, 2001, 2004) have shown that DBMS-level ICE, especially the referential constraints, is seldom enforced in real-world practice. Furthermore, many DBMSs (such as MySQL's default setting MySQL+MyISAM) do not fully support automatic checking of ICE.

We believe that a good database application should be "defensive" to prevent violations from happening. If the integrity constraints are missing from the database schema or if the DBMS is not configured properly to enable automatic enforcement of integrity constraints, we need to rely on application-level ICE. During long-term software evolution, the database schema, DBMS configuration or even the type of DBMS could be changed by different maintainers. Application-level ICE checking could help identify possible ICE-related problems and warn the maintainers when such changes occur. Furthermore, implementation of ICE at the application level can help detect errors in user input earlier, and therefore help provide better feedback to users and achieve better software usability.

To check the absence of ICE at the DBMS level is relatively easy. We just need to check whether the constraints are explicitly specified in the database schema and whether the DBMS is configured to enable automatic checking of integrity constraints. However, manually checking ICE at the application level is usually difficult, as the code base to be examined could be large. In Section 3, we propose a code pattern based approach for checking enforcement of integrity constraints at the application level.

2.2. An illustrative example of ICE at the application level

We now give an example to illustrate the concepts of ICE at the application level. Fig. 1 shows the conceptual model of a small customer-order management system, which involves three entities: `Customer`, `Order`, and `Item`. In relational database design, the attributes of the `Customer` table and their domain constraints are as follows: *custID* (not null, int type), *name* (not null, varchar type), *address* (not null, varchar type), and *phone* (not null, int type with 8 digits). Attribute *custID* is defined as the PK of table `Customer`, *orderID* as the PK of table `Order`, and *itemID* as the PK of table `Item`. As each order is associated with a customer, attribute *custID* in table `Order` is defined as a FK. Similarly, *orderID* in table `Item` is also a FK.

To enforce database integrity, a program needs to check whether the constraints are broken before performing any database operation. As an example, the PHP code in Fig. 2 implements input validation and key constraint enforcement. The code from line 6 to line 10 uses *if* structures to implement input validation, which checks the values of user input before they are accepted to the database. In line 18, a record with key value $*custID* is inserted

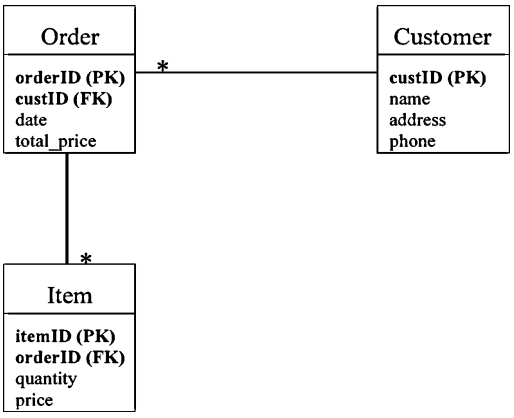

Fig. 1. The conceptual model of a database application.

```
1.  $custID = $_POST["custID"];
2.  $name = $_POST["name"];
3.  $addr = $_POST["address"];
4.  $phone = $_POST["phone"];
5.  ...
6.  if (!is_numeric($custID)) die('Wrong
      customer ID input');
7.  if($name==" ") die('Name cannot be
      empty!');
8.  if($addr==" ") die('Address cannot be
      empty!');
9.  if (!is_numeric($phone) || strlen($phone)!= 8)
10.    die('Invalid Phone Number!');
11. $sql = "SELECT * FROM Customer WHERE
      custID = ".$custID;
12. $result = mysql_query($sql);     //S
13. if ( $results == null) {         //P
14.    echo "Invalid Customer ID";
15.    exit();
16. }
17. $sql = "INSERT INTO Customer (custID,
      name, address, phone) VALUES(".$custID.
      "," .$name. ",".$addr. ",".$phone. ")";
18. mysql_query($sql);               //Q
```

Fig. 2. An example of PHP code implementing input validation and key constraint enforcement.

into table `Customer` via an SQL INSERT query. Before executing this SQL statement, code from line 11 to line 16 implements constraint enforcement for key *custID*. The predicate at line 13 checks whether the key value of $*custID* already exists in table `Customer` according to the result returned from the SQL SELECT query at line 12. If the check at line 13 is missing, constraint enforcement is not implemented at the application level and the key constraint may be violated.

Fig. 3 shows a code fragment that implements referential constraint enforcement at the application level. In line 13, a record with foreign key value $*custID* is to be inserted into table `Order` via an SQL INSERT query. Before executing this statement, code from line 6 to line 11 implements enforcement of the referential constraint.

```
1.  $orderID = $_POST["orderID"];
2.  $custID = $_POST["custID"];
3.  $date = $_POST["date"];
4.  $price = $_POST["total_price"];
5.  ...
6.  $sql = "SELECT * FROM Customer WHERE
      custID = ".$custID;
7.  $result = mysql_query($sql);       // S
8.  if ($results==null){               // P
9.     echo "Invalid Customer ID";
10.    exit();
11. }
12. $sql = "INSERT INTO Order (orderID,
      custID, date, total_price) VALUES (".$or-
      derID. "," .$custID. ",".$date. ",".$price. ")";
13. mysql_query($sql);                 // Q
```

Fig. 3. An example of PHP code implementing referential constraint enforcement for insertion/update.

The predicate at line 8 checks whether the FK value of $*custID* does exist in table `Customer` according to the result returned from the SQL SELECT query at line 7. If this check is missing, the referential constraint may be violated.

Fig. 4 shows an example of enforcing the referential constraint for deletion at the application level. Code at line 14 deletes a record with key value $*custID* in table `Customer` via an SQL DELETE query. To enforce database integrity, before executing line 14 we need to ensure that all records associated with $*custID* in table `Order` have been deleted (line 11). Note that, *orderID* is also a FK of table `Item`. To enforce the cascading nature of deletions, before executing line 11 we need to ensure that all records associated with $*custID* in table `Item` have been deleted (line 8).

```
1.  $sql = "SELECT * FROM Order WHERE
      custID = ".$custID;
2.  $result = mysql_query($sql);
3.  if ($results==null){
4.     echo "Invalid Customer ID";
5.     exit();
6.  }
7.  $sql ="DELETE from Item where orderID = "
      .$results['orderID'];
8.  mysql_query($sql);          // M
9.  ...
10. $sql ="DELETE from Order where custID = "
      .$custID;
11. mysql_query($sql);          // M
12. ...
13. $sql ="DELETE from Customer where custID
      = " .$custID;
14. mysql_query($sql);          // N
```

Fig. 4. An example of PHP code implementing referential constraint enforcement for deletion.

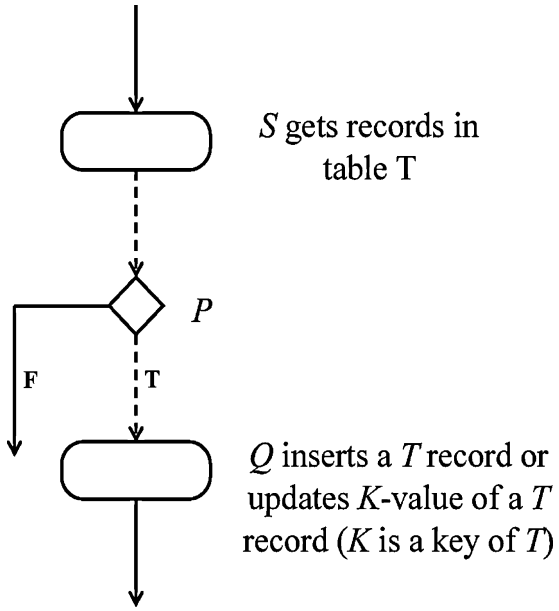

Fig. 5. The key constraint enforcement pattern.

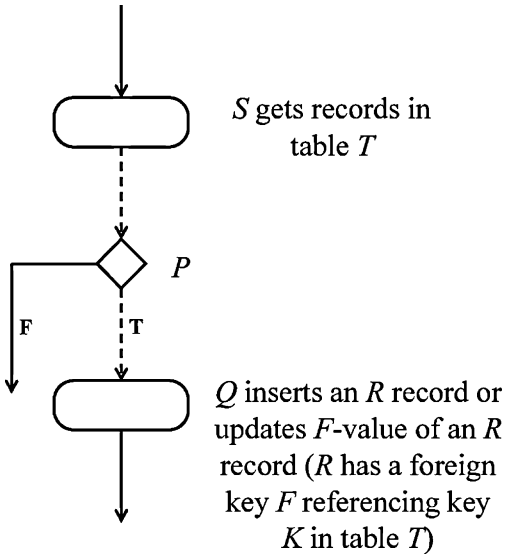

Fig. 6. The referential constraint enforcement pattern for insertion/update.

3. Automatic identification of application-level ICE

3.1. Code patterns

In this section, we introduce the discovered patterns for enforcing integrity constraints at the application level. These patterns are structural code patterns. Before moving further, we firstly define several basic terms that are used in this paper. We adopt the definition of the **control flow graph (CFG)** by Harrold et al. (1993) to represent a program.

In a CFG, node u **dominates** node w if and only if every path from the entry node to w contains u. Node w **postdominates** node u if and only if every path from u to the exit node contains w. Node y is **data dependent** on node x if and only if y is a successor of x and y refers to the data in x. Node y is **control dependent** on node x if and only if x has successors x' and x'' such that y post-dominates x' but y does not post-dominate x''. Furthermore, we say that node y is **transitively control (data) dependent** on node x if there is a sequence of nodes, $x = x_0, x_1, \ldots, x_n = y$, such that x_j is control (data) dependent on x_{j-1}, $1 \le j \le n$.

We also define "enforcement nodes" to denote predicate nodes that perform constraint enforcement. Let P be a predicate node in a CFG, T be a table in the database, and K be a column of T. If Q is a node that inserts or modifies T records such that the following conditions hold, we call P an **enforcement node** for K at Q:

(1) There exists a node S such that S dominates P, S gets T records and P is transitively data dependent on S.
(2) Q is transitively control dependent on P.
(3) No path from P to Q passes through any other node that inserts or updates T records.

We now describe four patterns that capture the structures of code implementing integrity constraint enforcement:

Pattern 1: Key constraint enforcement

Description: If K is a key of table T, then for each node Q in a CFG that inserts a T record or updates the K-value of a T record, there is an enforcement node P for K at Q.

Pattern 1 is illustrated in Fig. 5. The rationale is that, to enforce key constraints, whenever a record containing a key K is to be accepted to table T, a predicate P is needed to check whether a duplicate record with the same key value exists. An example of the pattern is given in Fig. 2, where K represents *custID* and T represents the `Customer` table. Code at line 18 represents Q, which inserts a new T record into the table. Code at line 13 represents P, which checks the key duplication. Code at line 12 represents S, which gets a T record. S is always executed before P, and Q is transitively control dependent on P.

Pattern 2: Referential constraint enforcement for insertion/update

Description: Let R and T be two tables in a database. Let F and K be an attribute of R and a key of T, respectively. If $R[F] \subseteq T[K]$ is a referential constraint, then for each node Q in a CFG that inserts an R record or modifies F-value of an R record, there is a predicate node P such that P is an enforcement node for K at Q.

Pattern 2 is illustrated in Fig. 6. The rationale is that, to enforce referential constraints, whenever a record containing a FK F is to be accepted to table R, a predicate P is needed to check whether the value of F exists as a PK value in the associated table T. An example of the pattern is given in Fig. 3, where F represents the FK *custID* of table `Order` (R) and K represents the PK *custID* of table `Customer` (T). Code at line 13 represents Q, which inserts a new record into table `Order`. Code at line 8 represents P, which checks the existence of the value of *custID* in table `Customer`. Code at line 7 represents S that gets R records. S is always executed before P, and Q is transitively control dependent on P.

Pattern 3: Referential constraint enforcement for deletion

Description: Let R and T be two tables in a database. Let F and K be an attribute of R and a key of T, respectively. If $R[F] \subseteq T[K]$ is a referential constraint, then for each node N in a CFG that deletes T records, there is a node M such that M deletes R records and N is a successor of M.

Pattern 3 is illustrated in Fig. 7. The rationale is that, to enforce referential constraints, whenever a record containing a key K is to be deleted from table T, a node that deletes the associated FK value F in table R should be executed first. An example of the pattern is given in Fig. 4, where F represents the FK *custID* of table `Order` and K represents the PK *custID* of table `Customer`. Code at line 14 (N) deletes a record from table `Customer`. Before that, code at line 11 (M) deletes all associated records in table `Order`. M is always executed before N. Note that, because of the cascading nature of referential constraints for deletions, records associated with the corresponding *orderID* in table `Item` should be deleted too (line 8, which is also an M node).

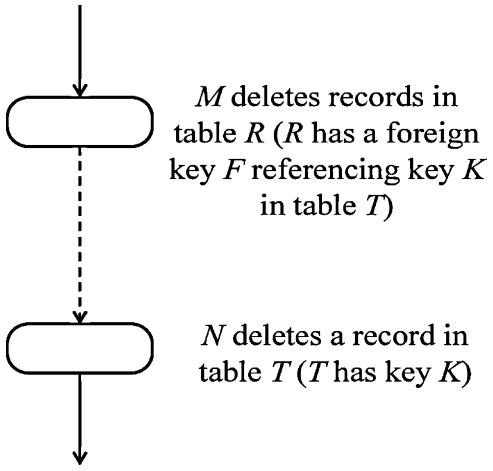

Fig. 7. The referential constraint enforcement pattern for deletion.

Pattern 4: Existence of input validation

Before describing the pattern, we first define "prime validation nodes", which denote predicate nodes that perform input validation.

In a CFG, let *I* be an input node that sets an input variable *V*, which is accepted to the database at node *Q*. We call a predicate node *P* a **prime validation node** for *V* at *Q* if the following properties hold:

(1) *Q* is transitively control dependent on *P*.
(2) *P* is transitively data dependent on *I*.
(3) *P* post-dominates *I*.
(4) No path from *P* to *Q* passes through any node at which *V* is accepted to the database.

Description: In a CFG of a transaction, for each input node *I* at which an input variable *V* is submitted, if there exists a node *Q* at which *V* is accepted to the database, there is a prime validation node *P* for *V* at *Q*.

Pattern 4 is illustrated in Fig. 8. The rationale is that, to enforce domain constraints, an input variable needs to be validated before being inserted or updated to the database. An example of the pattern is given in Fig. 2, where lines 1–4 contain four input nodes (*I*)

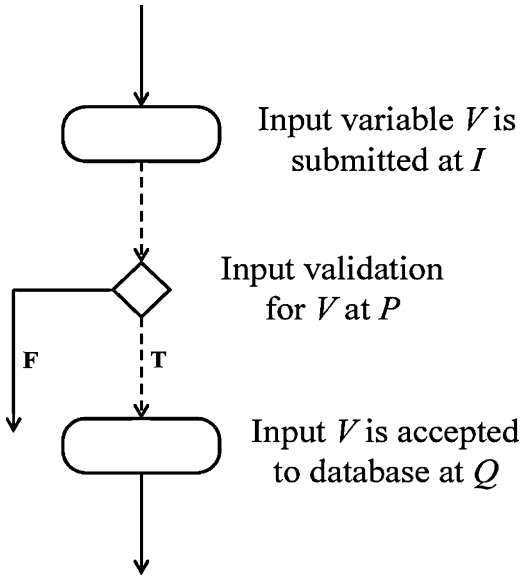

Fig. 8. The existence of input validation pattern.

```
Algorithm detectP1 (Π: a database application)
Output: ℑ: a set of tuples <T, K, Q, P>
begin
  ℑ = Ø;
  Ω=the set of basis paths in the CFG of Π;
  for each G in Ω do
    for each node Q in G do
      if Q inserts or modifies a T record and
        Q is not visited
      then for each key K of table T do
        if Q inserts or modifies K-value then
          Found = false;
          C = the set of predicate nodes on which
            Q is transitively control dependent;
          while C ≠ Ø or Found≠true do
            P = an element in C;
            C = C - {P};
            if P is the enforcement node for K at
              Q then
                Found = true;
                print the sub-path of G from the
                  entry to Q;
                ℑ = ℑ+{<T, K, Q, P>};
            endIf;
          endWhile;
          if Found == false then
            print the sub-path of G from the
              entry to Q;
            P = null;
            ℑ = ℑ+{<T, K, Q, P>};
          endIf;
        endIf;
      endFor;
      Q is visited;
      endIf;
    endFor;
  endFor;
end;
```

Fig. 9. An algorithm for detecting Pattern 1.

that take input variables (*V*) (i.e., $*custID*, $*name*, $*addr*, and $*phone*). Lines 6–10 (*P*) perform input validation for these variables before they are accepted to the database at line 18 (*Q*). Note that this pattern only specifies the existence of input validation, but it does not attempt to model all possible input validation scenarios (such as validation for security vulnerabilities).

3.2. Detecting enforcement of integrity constraints based on code patterns

From the patterns described in the previous sub-section, the structure of constraint enforcement can be verified automatically via static analysis. Violations of the patterns indicate that the program does not implement the corresponding constraint enforcement. A tool can be developed to scan the programs, perform automatic detection of violations of the patterns, and aid in determining whether the constraint enforcement is implemented at the application level. Figs. 9–12 show the algorithms

```
Algorithm detectP2 (Π: a database application)
Output: ℑ: a set of tuples <R, F, T, K, Q, P>
begin
  ℑ = Ø;
  Ω=the set of basis paths in the CFG of Π;
  for each G in Ω do
    for each node Q in G do
      if Q inserts or modifies an R record and Q is
       not visited
      then for each foreign key F of table R do
        if Q inserts or modifies F-value then
          Get tuple <T, K>, where R[F] ⊆ T[K]
           defines a referential constraint;
          Found = false;
          C=the set of predicate nodes on which
           Q is transitively control dependent;
          while C ≠ Ø or Found≠true do
            P = an element in C;
            C = C - {P};
            if P is the enforcement node for K
             at Q then
              Found = true;
              print the sub-path of G from the
               entry to Q;
              ℑ = ℑ+{<R, F, T, K, Q, P>};
            endIf;
          endWhile;
          if Found == false then
            print the sub-path of G from the
             entry to Q;
            P = null;
            ℑ = ℑ+{<R, F, T, K, Q, P>};
          endIf;
        endIf;
      endFor;
      Q is visited;
      endIf;
    endFor;
  endFor;
end;
```

Fig. 10. An algorithm for detecting Pattern 2.

```
Algorithm detectP3 (Π: a database application)
Output: ℑ: a set of tuples <R, F, T, K, M, N>
begin
  ℑ = Ø;
  for each Key in Π do
  Traverse the ER diagram of Π to obtain the lists
   of foreign keys that reference Key directly or
   indirectly, and store them as Propagation [key]
  endFor;
  Ω=the set of basis paths in the CFG of Π;
  for each G in Ω do
    for each node N in G do
      if N deletes a T record and N is not visited
      then for each foreign key F in
       Propagation [key] do
        Found = false;
        C=the set of nodes that dominate N;
        while C ≠ Ø or Found≠true do
          M=an element in C;
          C=C-{M};
          if M deletes R records then
            Found = true;
            print the sub-path of G from the
             entry to N;
            ℑ = ℑ+{<R, F, T, K, M, N>};
          endIf;
        endWhile;
        if Found == false then
          print the sub-path of G from the
           entry to N;
          N = null;
          ℑ = ℑ+{<R, F, T, K, M, N>};
        endIf;
      endFor;
      N is visited;
      endIf;
    endFor;
  endFor;
end;
```

Fig. 11. An algorithm for detecting Pattern 3.

for detecting patterns 1–4, respectively. Some explanations about the algorithms are as follows:

(1) Pattern 1: For each basis path (basis paths are a basic set of execution paths as proposed by McCabe (1976)) in the CFG and for each node in the basis path that inserts or modifies a key value of a record, the algorithm checks the CFG to determine whether the property stated in Pattern 1 is satisfied. The algorithm returns a set of tuples $\langle T, K, Q, P\rangle$, where T is a database table, K is a key of T, Q is the node that inserts or modifies K-values of T records, and P is an enforcement node for K at Q. If Pattern 1 is not detected for the tuple $\langle T, K, Q\rangle$, P is set to null and a violation case is detected.

(2) Pattern 2: For each basis path in the CFG and for each node in the basis path that inserts or modifies a foreign key of a record, the algorithm checks the CFG to determine whether the property stated in Pattern 2 is satisfied. The algorithm returns a set of tuples $\langle R, F, T, K, Q, P\rangle$, where $R[F] \subseteq T[K]$ defines a referential constraint. Q is the node that inserts or modifies the F-value of an R record, P is an enforcement node for K at Q. If Pattern 2 is not detected for the tuple $\langle R, F, T, K, Q\rangle$, P is set to null and a violation case is detected.

(3) Pattern 3: For each key in the database, the algorithm first traverses the entity-relationship (ER) diagram (i.e., the database conceptual model) and identifies the referential constraints in a cascading manner. Note that one key may correspond to more than one list of cascading referential constraints. For each basis path in the CFG and for each node in the basis path that deletes a record in a table that is referenced by a FK table, the algorithm then checks the CFG to determine whether the property stated in Pattern 3 is satisfied. The algorithm checks for each cascading

```
Algorithm detectP4 (Π: a database application)
Output: ℑ : a set of tuples <I, V, Q, P>
begin
  ℑ = Ø;
  Ω=the set of basis paths in the CFG of Π;
  Γ=the set of input variables of Π;
  for each G in Ω do
    for each node I in G do
      if I is an input node then
        for each input variable V submitted at I do
          for each node Q after I in G do
            if Q updates V to database and Q is
              not checked for V then
              Found = false;
              C=the set of predicate nodes on which
                Q is transitively control dependent;
              while C ≠ Ø or Found≠true do
                P=an element in C;
                C=C-{P};
                if P is a prime validation node for V
                  at Q then
                  Found=true;
                  print the sub-path of G from the
                    entry to Q;
                  ℑ = ℑ+{<I, V, Q, P>};
                endIf;
              endWhile;
              if Found == false then
                print the sub-path of G from the
                  entry to Q;
                P = null;
                ℑ=ℑ+{<I, V, Q, P>};
              endIf;
              Q is checked for V;
            endIf;
          endFor;
        endFor;
      endIf;
    endFor;
  endFor;
end;
```

Fig. 12. An algorithm for detecting Pattern 4.

referential constraint and returns a set of tuples $\langle R, F, T, K, M, N\rangle$, where $R[F] \subseteq T[K]$ defines a referential constraint. M is the node that deletes a T record, N is the node that deletes the associated R records. If Pattern 3 is not detected for the tuple $\langle R, F, T, K, M\rangle$, N is set to null and a violation case is detected.

(4) Pattern 4: For each basis path in the CFG and for each input variable at each input node in the basis path, the algorithm checks the CFG to determine whether the property stated in Pattern 4 is satisfied. The algorithm returns a set of tuples $\langle I, V, Q, P\rangle$, where V is the input variable submitted at node I, Q is the node that updates V to the database, and P is the primary validation node that performs the input validation for V at Q. If Pattern 4 is not detected for the tuple $\langle I, V, Q\rangle$, P is set to null and a violation case is detected.

For each case, the algorithms described in Figs. 9–12 output information about the paths from the program entry to the nodes that perform database operations. This information can be used for manual verification of the results.

3.3. Dealing with dynamic SQL statements

In some database applications, the SQL statements are hard-coded as an entire string in the program. In such a case, it is easy to know what database operation such an SQL statement performs. However, many database applications use SQL statements dynamically formed through string operations. In such a case, we use string analysis (Christensen et al., 2003; Minamide, 2005) to further analyze these dynamic SQL statements. For such an SQL statement, string analysis is used to acquire the context-free grammar representing all possible values of the SQL statement. We then use regular expressions to represent database operations that an SQL statement may perform. Thus, in order to check whether the dynamic SQL statement may perform a database operation, we check whether the intersection of the context-free grammar with the regular expression representing the database operation is empty. Such a check is similar to the check of cross-site scripting in Wassermann and Su (2008). As an example, the following regular expression represents the database operation of selecting attribute *attr* from table `tab`. For simplicity, we omit the parts for lower cases of letters, and use '∼' to represent the blank.

```
SELECT.*[,∼]ATTR[,∼].*FROM.*[,∼]TAB.*
```

3.4. PHPICE: a prototype tool

We have developed a prototype tool, called PHPICE, for checking enforcement of integrity constraints by detecting violations of the code patterns. We choose PHP based database applications as the targets of our tool, because PHP is one of the most widely used scripting languages for developing web-based database applications. PHPICE takes PHP database applications and constraint schema files as inputs, and checks whether the enforcement of integrity constraint is implemented at the application level.

PHPICE works in three steps: generating CFG, determining a basic set of paths from the entry to specific nodes, and searching for code patterns in the set. The generation of CFG is based on two open source tools: Pixy (Jovanovic et al., 2006a,b) – a tool for automatic detection of cross-site scripting (XSS) vulnerabilities in PHP programs,[1] and PHPXref (Huang et al., 2004) – a PHP Cross Referencing Documentation Generator.[2] Although originally developed for the purpose of checking security vulnerabilities, Pixy also supports general static program analysis including flow-sensitive, inter-procedural and context-sensitive control flow analysis. It first constructs a parse tree for a PHP program using the lexical analyzer JFlex[3] and the Java parser CUP,[4] and then transforms the parse tree into a linear intermediate representation in three-address code (Aho et al., 1986). Since Pixy lacks the ability to parse function calls defined in classes, we use PHPXref to complement Pixy. After CFGs are generated, PHPICE parses the CFGs, and implements the algorithms described in the previous sub-section. The input variables are identified by PHP constructs that read user input from HTTP forms, such as $_POST, $_GET, and $_REQUEST.

The constraint schema file processed by PHPICE is taken manually from design documentation, database schema files (such as .SQL files), or source code that create the database tables. It

[1] Pixy is available at: http://pixybox.seclab.tuwien.ac.at/pixy.
[2] PHPXref is available at: http://phpxref.sourceforge.net/.
[3] JFlex: The Fast Scanner Generator for Java, http://jflex.de.
[4] CUP: LALR Parser Generator in Java, http://www2.cs.tum.edu/projects/cup.

Table 1
The experimented PHP database applications.

System	#PHP files	Size	#Tables	#Keys	#FKs
OpenDocman	46	415 K	10	5	7
OpenBooking	42	252 K	11	11	9
InforCentral	139	1795 K	19	15	12
SchoolMate	63	241 K	15	19	27
Hotelmis	29	306 K	19	22	5
Catwin	119	415 K	20	18	8
phpEmailList	19	50 K	6	5	4
Phpay	121	859 K	29	29	41
Phpwims	67	494 K	6	6	1
Total	645	4827 K	135	130	114

contains information about tables, keys (including primary keys, unique keys and foreign keys) and integrity constraints among them. Primary keys that are generated automatically via the DBMS (marked as "auto_increment" in a database schema) are not processed since their enforcement is implemented by DBMS directly and SQL queries do not update these keys.

4. Experiments

4.1. Data collection

We conducted an empirical study on real-world PHP database (DB) applications to examine the status of integrity constraint enforcement and to evaluate our pattern based approach. We collected nine PHP DB applications from the open source web site *sourceforge.net*, including OpenDocMan (v1.2.4, a web-based document management system), OpenBooking (v0.62, a booking system), InforCentral (v1.2.6a, a membership management system), SchoolMate (v1.5.4, a school management system), hotelmis (v1.0, a hotel management system), Catwin (v0.7-1, an on-line accounting system), phpEmailList (v0.3, a newsletter system), Phpay (v2.02a, a web shop system), and Phpwims (0.4.2, a wine inventory management system). Table 1 shows the number of PHP files, the size of all files in bytes, the number of tables, the number of keys, and the number of foreign keys for each DB application.

One purpose of our empirical study is to understand whether ICE is implemented in the PHP DB applications. To achieve this purpose, we manually inspected the collected PHP applications. The other purpose of our empirical study is to evaluate our pattern based approach. To achieve this purpose, we ran PHPICE to automate the process of verifying ICE at the application level.

4.2. Manual inspection

ICE can be implemented at either the DBMS level or the application level. We first checked whether ICE is implemented at the DBMS level. When integrity constraints are automatically enforced by a DBMS, the DBMS ensures that the system is never left in an inconsistent stage. To achieve automatic enforcement of integrity constraints, the constraints must be explicitly specified during the creation of the database schema (e.g., in the .SQL file for MySQL). The DBMS needs to be told where the key is referenced so that it can keep the system consistent. For example, if one uses a foreign key in a table without actually declaring it as a foreign key, referential integrity cannot be enforced.

We checked the nine PHP applications and found that they all specify the primary keys in their database schemas. Furthermore, the DBMS's feature of "auto increment of unique key values" is widely used by the PHP applications to ensure the key constraint. Our results show that among the 95 unique keys, 89.47% is marked as "auto increment". Contrary to the key constraint, it is to our surprise that none of the nine PHP applications explicitly specifies

```
// OpenBooking.sql
CREATE TABLE tbl_Activities (
  ID int(10) unsigned NOT NULL
    auto_increment,
  Name varchar(50) NOT NULL default '',
      ...
  PRIMARY KEY (ID)
) TYPE=MyISAM;
CREATE TABLE tbl_Members (
  ID int(10) unsigned NOT NULL
    auto_increment,
      ...
  BirthDate date NOT NULL default
    '0000-00-00',
  PRIMARY KEY (ID)
) TYPE=MyISAM;
CREATE TABLE tbl_Reservations (
  ID int(10) unsigned NOT NULL
    auto_increment,
  MemberID int(10) unsigned NOT NULL
    default '0',
  ActivityID int(10) unsigned NOT NULL
    default '0',
      ...
  PRIMARY KEY (ID)
) TYPE=MyISAM;
```

Fig. 13. A fragment of the database schema of the OpenBooking system.

the foreign keys in their database schemas. The referential constraint enforcement is missing at the DBMS level. As an example, Fig. 13 shows a fragment of the database schema of the OpenBooking system. A primary key is specified for all three tables (i.e., `tbl_Activities`, `tbl_Members` and `tbl_Reservations`). However, clearly as implied by the attribute names (and confirmed by the program logic), attributes *MemberID* and *ActivityID* of table `tbl_Reservations` should be foreign keys. Failing to specify the foreign keys prevents the DBMS from handling the referential constraint automatically.

For the constraints that are not automatically enforced by the DBMS, we checked whether they are enforced at the application level. We first manually created the constraint schema file based on the database schema, source code and/or design documents. For example, in Fig. 13 the PK of table `tbl_Activities` serves as a FK of table `tbl_Reservations`. We then manually examined all SQL INSERT, UPDATE, and DELETE statements to check whether ICE is implemented before database operations. The results are shown in Table 2. We have inspected 668 SQL statements and identified 465 instances of ICE absence.

Figs. 14–16 give some actual examples of absence of ICE in PHP programs. Fig. 14 shows an example of the absence of key constraint enforcement found in the Hotelmis system. Attribute *loginname* is defined as a unique key (NOT NULL) in table `users`. However, in *admin.php*, a new record is inserted into table `users` without checking the unique key constraint on *loginname*. This code fragment does not exhibit the structure defined in pattern 1 and may lead to duplicate key values. We also found non-existence of input validation for variables $*fname*, $*sname*, and $*loginanme* (whose corresponding database attributes are all defined as NOT NULL), as these variables are accepted to table `users` without any validation. However, the value of variable $*reports* is validated.

Fig. 15 shows an example of violation of the referential constraint enforcement for insert/update operations in the Open-

Table 2
The absence of integrity constraint enforcement in studied PHP applications. (Total SQL represents the total number of SQL INSERT, DELETE, UPDATE statements; KCE represents Key Constraint Enforcement; RCE1 represents Referential Constraint Enforcement for insertion and update; RCE2 represents Referential Constraint Enforcement for deletion; and IV represents input validation.)

System	Total SQL	KCE	RCE1	RCE2	IV
OpenDocman	51	1	11	3	36
OpenBooking	41	0	15	8	38
InforCentral	117	0	14	5	0
SchoolMate	79	0	14	26	74
Hotelmis	25	3	12	0	31
Catwin	151	6	22	5	2
phpEmailList	18	0	5	2	5
Phpay	119	0	65	59	0
Phpwims	67	2	0	1	0
Total	668	12	158	109	186

```
//  in admin.php (Hotelmis), lines 51-80:
1.  $fname=$_POST["fname"];
2.  $sname=$_POST["sname"];
3.  $loginname=$_POST["loginname"];
4.  ...
5.  $reports=(empty($_POST["reports"]))?
    0:$_POST["reports"];
6.  $sql="INSERT INTO users (fname, sname,
    loginname, ..., reports) VALUES('$fname',
    '$sname', '$loginname', ..., $reports)";
7.  $results=mkr_query($sql,$conn);
```

Fig. 14. An example of absence of key constraint enforcement and input validation.

Booking system. Attribute *ResourceGroupID* is a foreign key of table `tbl_Activities` that references the primary key of table `ResourceGroup`. Therefore, to enforce the referential constraint, before inserting a record into table `tbl_Activities` we need to check whether the value of user input *resourceGroupID* does exist in the referenced table. However, this code fragment does not implement ICE and may lead to violation of the referential constraint. Furthermore, the code in Fig. 15 also reveals the existence/absence of input validations. User input *name* is validated before accepted into the database. However, user inputs *description*, *resourceGroupID*, and *sectionCount* (whose corresponding database

```
//  in Activities.php (OpenBooking), lines 46-58:
1.  $act = "add";
2.  if ($_REQUEST['name'] == " ") {
3.     $message = "CORRECTION: You MUST
         enter a name for this activity.";
4.  } else {
5.     $activityID = mysql_send("INSERT
         INTO tbl_Activities (Name, Description,
          ResourceGroupID, SectionCount)
         VALUES ('". $_REQUEST['name'].
         "','". $_REQUEST['description']. "','".
         $_REQUEST['resourceGroupID']. "','".
          $_REQUEST['sectionCount']. "')");
6.     $act = " ";
7.     $message="NOTE:Activity successfully
         added to system";
8.  }
```

Fig. 15. An example of absence of referential constraint enforcement for insert/update operations and input validation.

```
//  in member.php (OpenBooking), lines 222-230:
1.  if ($act == "remove") {
2.     if ($_REQUEST['memberID'] != " ") {
3.        mysql_send("DELETE FROM
            tbl_Members WHERE ID=
            {$_REQUEST['memberID']}");
4.        $message="NOTE: Member successfully
            removed.";
5.     } else {
6.     $message = "ERROR: Please select a
         Member.";
7.     }
8.  ...}
```

Fig. 16. An example of the absence of referential constraint enforcement for deletion.

attributes are all defined as NOT NULL) are directly accepted to the database without any validation.

Fig. 16 shows a violation of referential constraint enforcement for delete operations found in the OpenBooking system. The code deletes a record in table `tbl_Members` where *ID=memberID*. However, further analysis finds that the primary key *ID* is also associated with a foreign key in table `tbl_Reservations`. Therefore, to enforce the referential constraint, before deleting a record in table `tbl_Members`, we need to delete all associated records in the referencing table (i.e., `tbl_Reservations`). The code in Fig. 16 does not implement ICE and may thus break the referential constraint.

The experimental results show that the enforcement of integrity constraints, especially the enforcement of referential constraints and domain constraints, is often overlooked in programming practice. This problem could potentially result in inconsistent data, runtime exceptions, or security flaws. Even if the ICE is implemented at the DBMS level, the applications are still error-prone during software evolution when the DBMS could be reconfigured or replaced. The absence of ICE at the application level also decreases the usability of the applications, because end-users are only aware of the data errors after they occur. To improve the dependability and usability of database applications, programmers should pay more attention to enforcement of integrity constraints.

4.3. Automatic detection of absence of ICE at the application level

Manual identification of ICE in programs is tedious and time consuming. Our code-pattern based prototype tool PHPICE can help automate this task. We ran PHPICE on the nine PHP applications to detect absence of ICE at the application level. To evaluate the accuracy of PHPICE, we use Recall and Precision, which are metrics widely adopted in information retrieval (Baeza-Yates and Ribeiro-Neto, 1999) and data mining (Witten and Frank, 2005). The Recall defines the rate of true ICE absence cases detected by our tool in comparison to the total number of true ICE absence cases. The Precision relates to the number of true ICE absence cases detected by our tool in comparison to the total number ICE absence cases reported by our tool. The values of the Recall and the Precision are from 0 to 1. A good detection tool should achieve both high Recall values and high Precision values.

For each case of pattern violation reported by PHPICE, we checked whether it indicates an actual ICE absence by comparing the results with those obtained from manual inspection. We then computed the Recall and the Precision values for each application. For all the applications, the prototype tool is able to detect all ICE absences that are discovered by manual inspection. Therefore the Recall values are all 100%. Our tool does report a few false positive cases, as indicated by the Precision values in Table 3. In total, based

Table 3
The precision of automatic detections.

System	KCE	RCE1	RCE2	IV
OpenDocman	100%(1/1)	91.67%(11/12)	100%(3/3)	100%(36/36)
OpenBooking	N/A(0/0)	88.24%(15/17)	100%(8/8)	100%(38/38)
InforCentral	0%(0/1)	87.5%(14/16)	83.33%(5/6)	N/A(0/0)
SchoolMate	N/A(0/0)	87.5%(14/16)	92.86%(26/28)	98.67%(74/75)
Hotelmis	100%(3/3)	100%(12/12)	N/A(0/0)	100%(31/31)
Catwin	100%(6/6)	91.67%(22/24)	100%(5/5)	66.67%(2/3)
phpEmailList	N/A(0/0)	83.33%(5/6)	100%(2/2)	100%(5/5)
Phpay	N/A(0/0)	97.0%(65/67)	100%(59/59)	N/A(0/0)
Phpwims	100%(2/2)	N/A(0/0)	100%(1/1)	N/A(0/0)
Total	92.31%(12/13)	92.94%(158/170)	97.32%(109/112)	98.93%(186/188)

on pattern 1 our tool can detect absences of key constraint enforcement with precision 92.31% (12 out of 13); based on pattern 2 our tool can detect absences of referential constraint enforcement (for insertion/update) with precision 92.94% (158 out of 170); based on pattern 3 our tool can detect absences of referential constraint enforcement (for deletion) with precision 97.32% (109 out of 112); based on pattern 4 our tool can detect absences of input validation with precision 98.93% (186 out of 188). All precision values are satisfactory. The results confirm that our code-pattern based approach is effective in discovering absences of ICE at the application level.

Figs. 14–16 show some code examples of true positive cases. Our tool can successfully detect all these cases. We now examine the false positive cases. Many cases are related to the generation of keys using automatically retrieved IDs. Considering the example given in Fig. 17, lines 4 and 5 (about an SQL INSERT query), the value of primary key *per_ID* of table `person_custom` is assigned with the return value of PHP function *mysql_insert_id*(), which retrieves the ID generated for an "auto_increment" attribute by the previous INSERT query. In this example, the PK of table `person_per` is automatically increased by the DBMS, therefore the values of PK *per_ID* is always increasing and unique. Although this example does not match pattern 1, it enforces the key constraint. However, we should point out that this is not good programming practice as function *mysql_insert_id*() only acts on the last performed query and must be called immediately after that query. Therefore, the example in Fig. 17 may experience maintenance problems when possible new queries are inserted between the existing queries.

Fig. 18 shows another example of false positive cases of checking input validation. Our tool considers the input variable $_*POST*[*date*] as not validated. However, the variable is actually assigned with the current date value via a PHP library function named *date*(). Hence variable $_*POST*[*date*] no longer stores the user input from the HTTP form and it should not be considered as a violation of input validation. However, we note that it is bad programming practice as input variables should not be used to store local values.

4.4. Current limitations

Our approach and prototype tool have limitations too. Currently PHPICE relies on the analysis of SQL queries to recognize the proposed patterns. In recent years, some database programming techniques such as the object-relational mapping (ORM) framework have been adopted for mapping objects to relational databases (Ambler, 2003). In ORM, SQL queries are often hidden from the application developer. Our approach cannot be directly applied to programs written in ORM (but fundamentally, our approach is still applicable as the underlying ORM implementations still use SQL queries).

Another limitation is that the current prototype tool can only analyze PHP programs. We believe that the ICE problem does not pertain to a specific programming language. The problem is due to programmers' ignorance or bad habits. Our approach can be applied to DB applications written in other languages (such as Java or Python), as the proposed patterns model general solutions for common DB problems. Thus, it is also possible to extend our tool to recognize the four ICE-related code patterns in applications written in other languages via static analysis. The current tool also has limited power in static analysis of PHP programs. It does not support the analysis of alias of pointers and complex data structures. In the future we will explore better alias and literal analysis techniques to handle references in PHP programs.

5. Related work

Data quality issues are important for database applications. Many database researchers have proposed various approaches that aim to automatically and efficiently detect data inconsistencies at the DBMS level. For example, Fan et al. (2008) proposed a class

```
//  in familyEditor.php (InforCentral),
//  lines 198-203:
1.  RunQuery("LOCK TABLES person_per
    WRITE, person_custom WRITE");
2.  $sSQL = "INSERT INTO person_per
    (per_FirstName, per_MiddleName,
    per_LastName, per_fam_ID, per_fmr_ID,
    ...) VALUES (...)";
3.  RunQuery($sSQL);
4.  $sSQL = "INSERT INTO person_custom
    (per_ID) VALUES (" . mysql_insert_id() . ")";
5.  RunQuery($sSQL);
6.  RunQuery("UNLOCK TABLES");
```

Fig. 17. False positive example one.

```
//  in ManageAnnouncements.php (Schoolmate),
//  lines 4-10:
1.  if($_POST["addannouncement"] == 1 &&
    $_POST["title"] != "" && $_POST["message"]
    != " ") {
2.    $_POST["date"] = date("Y-m-d");
3.    $query = mysql_query("INSERT INTO
      schoolbulletins VALUES('',
      '$_POST[title]','$_POST[message]',
       '$_POST[date]')") or die("Unable to
      insert new announcement - ". mysql_error());
4.  }
```

Fig. 18. False positive example two.

of integrity constraints for relational databases, referred to as the conditional functional dependencies (CFDs). They studied the use of CFDs in data cleaning and developed techniques for detecting CFD violations in SQL statements. Fan (2008) also proposed a concept of data dependencies, called matching dependencies (MDs), for improving data quality including the detection of the violation of integrity constraints. Song and Chen (2009) proposed metrics for discovering MDs with minimum confidence and support requirements. Jurk and Balaban (2001) proposed to represent the task of integrity constraint enforcement using extended rules and dependency graphs, and developed an optimization technique for efficient implementation of ICE.

Although ICE is important for ensuring quality of database applications, there are still relatively less research efforts that are dedicated to checking ICE in the area of software engineering. Some notable work is performed by Blaha (2001), who studied about 50 database applications and found out that 90% of them did not support ICE at the DBMS level. Blaha (2004) further proposed to reverse engineer a vendor's database as part of an evaluation to determine overall software quality before buying a product. Our work shows that ICE is often missing at the application level too. We proposed a code pattern based approach to detecting the absence of ICE at the application level. Combining Blaha's approach and ours, we may achieve a better evaluation of the overall quality of a database application. Dasgupta et al. (2009) described a framework that analyzes database applications based on ADO.NET and showed how the framework can be used for a variety of analysis such as SQL injection detection, workload, etc. For data integrity, they only check domain constraints such as "if a variable is less than 0". The key and referential constraints are not addressed. Through a study of a large commercial system, Maule et al. (2008) showed that changes to database schemas have impact on database applications. Their results confirmed that DBMSs and applications could mismatch during software evolution. They proposed approaches for predicting the impact of schema changes upon object-oriented applications. In our study, we focus on checking enforcement of integrity constraints. Our approach can be used to check whether database schema changes may lead to violations of integrity constraints. Thiran et al. (2006) also observed that incorporating a new program in a legacy database application can produce integrity mismatch. They noted that "neither the legacy DBMS (too weak to address integrity issues correctly) nor the new program (that relies on data server responsibility) correctly cope with data integrity management". They proposed a wrapper based architecture to reconciliate the mismatches. Unlike their approach, our proposed approach is based on code patterns, without forcing application developers to employ certain software architecture.

In recent years, a large number of studies have been conducted to detect bugs and problems in programs based on code patterns. For example, FindBugs (FindBugs, 2011; Hovemeyer and Pugh, 2004) and PMD (2011) can analyze Java code and report warnings based on hundreds of bug patterns such as null pointer dereference, read of unwritten fields, etc. Li and Zhou (2005) used a frequent itemset mining technique to identify similar code patterns and violations of programming rules. Livshits and Zimmermann (2005) found application-specific bug patterns by mining software revision histories. Violations of these patterns are responsible for a multitude of errors. Unkel and Lam (2008) proposed to infer whether a field can be legally declared as final in a program by examining the usage pattern of the field in the code. This information can then be used for bug detection. All these studies revealed that many types of bugs and problematic code in programs do exhibit certain patterns, and therefore by mining these patterns, we can discover many previously unknown bugs and problems statically. The patterns discovered by the related work are applicable to general programs. In this paper, we show that we can also discover code patterns for a specific domain. We focus on the domain of database applications, where we discover four constraint enforcement related code patterns. We also formally describe the identified patterns in terms of program analysis. In our prior work, we also discovered many different types of code patterns. For example, in Ngo and Tan (2007), we described several patterns for infeasible paths in Java programs. We also developed tools to detect a large number of infeasible paths through recognizing these patterns. In Liu and Tan (2008, 2009), we proposed approaches for testing input validation in Java-based web applications via identification of patterns. The work presented in this paper is similar in spirit to our prior work, but focuses on a different problem.

There is also much work on detecting security vulnerabilities for PHP programs. For example, Xie and Aiken (2006) proposed an inter-procedural static analysis algorithm for detecting SQL injection vulnerabilities. They employed a three-tiered architecture to handle unique features of PHP such as dynamic typing and code inclusion. They performed experiments on six PHP applications and found 105 security vulnerabilities. Jovanovic et al. (2006a,b) developed Pixy, an open source tool for detecting cross-site scripting vulnerabilities in PHP programs. They discovered 15 vulnerabilities in three PHP applications. Wassermann and Su (2008) also presented a static analysis approach for finding cross-site scripting (XSS) vulnerabilities using tainted information flow with string analysis. They believed that the root cause of the XSS is weak validation and checked whether an untrusted input can invoke a browser's JavaScript interpreter. Similarly, our work also detects problematic code in PHP programs via static analysis. The underlying techniques used, such as data and control flow analysis and string analysis, have much in common. In fact, our tool is developed based on Pixy. But our work focuses on a different domain (i.e., the database application domain) and addresses the problem of integrity constraint enforcement.

6. Conclusions

Integrity constraint enforcement (ICE) is an important feature of database applications. Breaking these constraints may lead to loss of data dependencies or even corruption of the database. It is important for database applications to perform ICE to ensure data consistency.

In this paper, we have investigated the status of ICE using nine open-source PHP database applications. The results show that ICE, especially the enforcement of referential constraints, does not receive enough attention in programming practice. We have proposed a pattern based approach for checking the absence of ICE at the application level. We have described four patterns that characterize the structure of code implementing ICE. Violations of these patterns indicate the missing of constraint enforcement source code. We have also developed a tool called PHPICE to detect the absence of the patterns in PHP programs. Using the tool we have successfully discovered many previously unknown violations of constraint enforcement in these applications with high accuracy.

In the future, we plan to apply the proposed approach to industrial database applications to further evaluate its effectiveness. We will address the limitations described in the paper and further improve our approach. We will also explore test case generation techniques based on the proposed patterns and investigate whether the verification of ICE can be performed by dynamic analysis.

Acknowledgments

We thank the developers of Pixy and PHPXref for releasing their tools, which help a lot in the development of our prototype tool

PHPICE. We thank undergraduate students Chongdi Huang and Fusheng Wang for their help in data collection, initial tool development, and result verification. We also thank the anonymous reviewers for the detailed comments, which have helped improve this paper. The authors from Tsinghua University are supported by the NSFC grant 61073006, National HGJ Project 2010ZX01045-002-3, and the Tsinghua University research project 2010THZ0. The authors from Peking University are supported by the 973 Program of China No. 2009CB320703 and the Science Fund for Creative Research Groups of China No. 60821003.

References

Ramakrishnan, R., Gehrke, J., 2000. Database Management Systems, 2nd edition. McGraw-Hill.
Silberschatz, A., Korth, H., Sudarshan, S., 2005. Database System Concepts. McGraw-Hill.
Blaha, M., 2001. A retrospective on industrial database reverse engineering projects. In: Proceedings of Working Conference on Reverse Engineering, pp. 136–147.
FindBugs, 2011. http://findbugs.sourceforge.net, May.
Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. In: Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Companion), pp. 132–136.
PMD, 2011. http://pmd.sourceforge.net/, May.
Blaha, M., 2004. A copper bullet for software quality improvement. IEEE Computer 37 (2), 21–25.
Harrold, M., Malloy, B., Rothermel, G., 1993. Efficient construction of program dependence graphs. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 160–170.
McCabe, T., 1976. A complexity measure. IEEE Transactions on Software Engineering 2 (4), 308–320.
Christensen, A., Møller, A., Schwartzbach, M., 2003. Precise analysis of string expressions. In: Proceedings of the International Static Analysis Symposium, pp. 1–18.
Minamide, Y., 2005. Static approximation of dynamically generated web pages. In: Proceedings of International Conference on World Wide Web, pp. 432–441.
Wassermann, G., Su, Z., 2008. Static detection of cross-site scripting vulnerabilities. In: Proceedings of the International Conference on Software Engineering, pp. 171–180.
Jovanovic, N., Kruegel, C., Kirda, E., 2006a. Pixy: a static analysis tool for detecting web application vulnerabilities. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 258–263.
Jovanovic, N., Kruegel, C., Kirda, E., 2006b. Precise alias analysis for static detection of web application vulnerabilities. In: Proceedings of ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pp. 27–36.
Huang, Y., Yu, F., Hang, C., Tsai, C., Lee, D., Kuo, S., 2004. Verifying web applications using bounded model checking. In: Proceedings of IEEE/IFIP Conference on Dependable Systems and Networks, pp. 199–208.
Aho, A.V., Sethi, R., Ullman, J.D., 1986. Compilers: Principles, Techniques, and Tools. Addison-Wesley.
Baeza-Yates, R., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley.
Witten, I., Frank, E., 2005. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementation, 2nd edition. Morgan Kaufmann.
Ambler, S., 2003. Agile Database Techniques: Effective Strategies for the Agile Software Developer. Wiley.
Fan, Wenfei, Geerts, Floris, Jia, Xibei, Kementsietsidis, Anastasios, 2008. Conditional functional dependencies for capturing data inconsistencies. ACM Transactions on Database Systems 33 (2), doi:10.1145/1366102.1366103, Article 6 (June 2008), 48 pages.
Fan, W., 2008. Dependencies revisited for improving data quality. In: Proceedings of the International Symposium on Principles of Database Systems, pp. 159–170.
Song, S., Chen, L., 2009. Discovering matching dependencies. In: Proceedings of the International Conference on Information and Knowledge Managment, pp. 1421–1424.
Jurk, S., Balaban, M., 2001. Improving integrity constraint enforcement by extended rules and dependency graphs. In: Proceedings of International Conference on Database and Expert Systems Appliactions, pp. 501–516.
Dasgupta, A., Narasayya, V., Syamala, M., 2009. A static analysis framework for database applications. In: Proceedings of the International Conference on Data Engineering, pp. 1403–1414.
Maule, A., Emmerich, W., Rosenblum, D., 2008. Impact analysis of database schema changes. In: Proceedings of the International Conference on Software Engineering, pp. 451–460.
Thiran, P., Hainaut, J., Houben, G., Benslimane, D., 2006. Wrapper-based evolution of legacy information systems. ACM Transactions on Software Engineering and Methodology 15 (4), 329–359.
Li, Z., Zhou, Y., 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Proceedings of the Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 306–315.
Livshits, B., Zimmermann, T., 2005. DynaMine: finding common error patterns by mining software revision histories. In: Proceedings of the Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 296–305.
Unkel, C., Lam, M., 2008. Automatic inference of stationary fields: A generalization of java's final fields. In: Proceedings of the International Symposium on Principles of Programming Languages, pp. 183–195.
Ngo, M.N., Tan, H.B., 2007. Detecting large number of infeasible paths through recognizing their patterns. In: Proceedings of the Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 215–224.
Liu, H., Tan, H.B., 2008. Testing input validation in Web applications through automated model recovery. The Journal of Systems and Software 81 (2), 222–233.
Liu, H., Tan, H.B., 2009. Covering code behavior on input validation in functional testing. Information and Software Technology 81 (2), 546–553.
Xie, Y., Aiken, A., 2006. Static detection of security vulnerabilities in scripting languages. In: Proceedings of USENIX Security Symposium, pp. 179–192.

Hongyu Zhang is an Associate Professor at the School of Software, Tsinghua University, Beijing, China. He received the PhD degree in computer science from the School of Computing, National University of Singapore in 2003, the MS degree in communication and networks from Nanyang Technological University, Singapore in 1998. Before joining Tsinghua University in 2006, he was a lecturer at the School of Computer Science and Information Technology, RMIT University, Australia. His research is mainly in the area of software engineering, in particular, software quality, software maintenance, empirical software engineering, and software reuse. He has published more than 40 research papers in international journals and conferences proceedings. He is a member of the program committee of many conferences, and an invited reviewer for many software engineering journals.

Hee Beng Kuan Tan is an Associate Professor with the Information Engineering Division in the School of Electrical and Electronic Engineering, Nanyang Technological University. He received his B.Sc. (1 Hons) in Mathematics in 1974 from the Nanyang University (Singapore), and his M.Sc. and Ph.D. degrees in Computer Science from the National University of Singapore in 1989 and 1996 respectively. His current research interest is in software security, software analysis and testing. Contact him at ibktan@ntu.edu.sg.

Lu Zhang is a Professor in the Institute of Software, School of Electronics Engineering and Computer Science, Peking University, P.R. China. He received both PhD and BSc in Computer Science from Peking University in 2000 and 1995 respectively. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, UK. He was a program co-chair of SCAM2008 and in the program committee of several conferences. He has been on the editorial boards of *Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification and Reliability*. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse and component-based software development, and service computing.

Xi Lin received his B.Eng. degree in software engineering from the School of Software, Tsinghua University in 2010. His research interests include software verification and program analysis.

Xiaoyin Wang received his BS degree from the Department of Computer Science at Harbin Institute of Technology in 2006. Since September 2006, he has been a PhD student in the School of Electronic Engineering and Computer Science at Peking University. His research interests include software maintenance, software mining, and software refactoring.

Chun Zhang received his MS degree in computer science from Peking University in 2011. His research interests include program analysis and testing.

Hong Mei is a professor at the School of Electronics Engineering and Computer Science, Peking University. He received his Ph.D. in Computer Science from Shanghai Jiaotong University in 1992. He is a senior member of IEEE. He was a program co-chair of many software conferences, including COMPSAC2005, QSIC2006, COMPSAC2007, ICSR2008, ICSM2008, and ICWS2008. He also serves on the editorial board of *IEEE Transactions on Service Computing and International Journal of Web Services Research*. His current research interests include software engineering environment, software reuse and component technology, distributed object technology, and programming language.