Automatically Detecting Integrity Violations In Database-Centric Applications

Boyang Li
College of William and Mary
Williamsburg, VA 23185
Email: boyang@cs.wm.edu

Denys Poshyvanyk
College of William and Mary
Williamsburg, VA 23185
Email: denys@cs.wm.edu

Mark Grechanik
University of Illinois at Chicago
Chicago, IL 60607
Email: drmark@uic.edu

***Abstract*—Database-centric applications (DCAs) are widely used by many companies and organizations to perform various control and analytical tasks using large databases. Real-world databases are described by complex schemas that oftentimes contain hundreds of tables consisting of thousands of attributes. However, when software engineers develop DCAs, they may write code that can inadvertently violate the integrity of these databases. Alternatively, business analysts and database administrators can also make errors that lead to integrity violations (semantic bugs). To detect these violations, stakeholders must create assertions that check the validity of the data in the rows of the database tables. Unfortunately, creating assertions is a manual, laborious and error-prone task. Thus, a fundamental problem of testing DCAs is how to find such semantic bugs automatically.**

We propose a novel solution, namely DACITE, that enables stakeholders to automatically obtain constraints that semantically relate database attributes and code statements using a combination of static analysis of the source code and associative rule mining of the databases. We rely on SAT-solvers to validate if a solution to the combined constraints exists and issue warnings on possible semantic bugs to stakeholders. We evaluated our approach on eight open-source DCAs and our results suggest that semantic bugs can be found automatically with high precision. The results of the study with developers show that warnings produced by DACITE are useful and enable them to find semantic bugs faster.

I. Introduction

Database-centric applications (DCAs) are widely used by many companies and organizations to perform various control and analytical tasks, e.g., computing decision rules using data analysis. DCAs are often written using some popular programming languages like Java, and they access and manipulate data from databases. These databases in practice are quite large; they contain hundreds of tables consisting of thousands of attributes [62]. Business analysts and database administrators create and maintain logical and physical schemas of databases that include sophisticated checks – assertions that check validity of the data in the database tables [17]. Creating these assertions is an intellectually laborious process that requires business analysts to understand deeply the semantics of the data. For example, a simple assertion that specifies that males do not give birth is often not encoded explicitly in databases of insurance companies based on our examination of several commercial database schemas; many assertions can be inferred only by analyzing the data in these databases. Given that it is expensive to create and encode assertions, many database schemas do not contain them, and this situation may result in incorrect data put into databases, thus, violating the integrity of these databases.

When software engineers develop DCAs they write code that pulls data from the databases, performs computations on this data and updates the data in these databases. Users of these DCAs can violate the integrity of databases simply by entering incorrect data. For example, a doctor may make a mistake in the gender of a patient who gave birth when using a medical coding DCA to submit billing information for this patient. An insurance company may later retrieve this patient's data and compute a reimbursement for the procedure. Of course, a simple check based on the assertion $\forall p, \text{Patient}(p) \wedge \text{Gender}(p) = M \rightarrow \neg\text{Treatment}(\text{Birth})$ can solve this problem. Moreover, a flight booking system should not allow selling more seats for a flight than the plane's capacity. Unfortunately, manually creating thousands of assertions is infeasible for complex database schemas, since programmers must obtain deep understanding of the semantics of the data.

In addition, programmers can make mistakes while developing DCAs by incorrectly using the values of database attributes. Based on our participation in large-scale projects, we observed that making incorrect assumptions in code about schemas is not that uncommon. Given that many schemas contain thousands of attributes and evolve rapidly in practice, it is easy to make mistakes about the semantics of the data that these attributes hold and about relations among these attributes. The source of errors often lies in the complexity of schemas that business analysts and database administrators create and maintain leading them to formulate incorrect assertions. Modern databases have complex semantics and mastering them requires a steep learning curve.

Often programmers lack the knowledge of the impact caused by changing the code of some DCA on other DCAs that interoperate using databases. This lack of knowledge is an effect of the Curtis' law that states that application and domain knowledge is thinly spread and only one or two team members may possess the full knowledge of a software system [25]. The effect of this law combined with the difficulty of comprehending large-scale database schemas and the high complexity of the source code of DCAs results in integrity violations that are very difficult to localize.

The other source of errors is the disparity in evolving database schemas and the source code of DCAs. Business analysts and system administrators usually maintain schemas, and programmers maintain the source code of DCAs. If a database administrator modifies some schemas without informing all programmers whose DCAs are affected by this change, then some DCAs will keep using databases according to the obsolete schemas resulting in integrity violations.

We propose a novel solution for *DcA Constraints resolutIon and TEsting (DACITE)* that enables stakeholders to automatically obtain assertions that semantically relate database attributes and codes. That is, after annotating a small number of expressions and statements in the source code that obtain values of attributes from the databases, a static analysis is per-

formed to obtain some constraints that relate these attributes. On the database side, we apply *associative rule mining (ARM)* to obtain constraints directly from the data that relate different attributes. Next, we rely on SAT-solvers to determine if a solution exists for the combined constraints. If no such solution is found, then DACITE marks these constraints as potential semantic errors and issues warnings to stakeholders. Our paper makes the following noteworthy contributions;

- We propose a novel solution to the problem of detecting potential integrity violations in DCAs. To the best of our knowledge, this is the first solution to this difficult and important problem;
- We implemented DACITE, evaluated it on several open-source DCAs and determined that it is effective and efficient in finding certain classes of integrity violations;
- We conducted a study with developers who used DACITE for detecting semantic bugs. The results demonstrate that developers find that DACITE warnings are useful and they can find semantic bugs efficiently;
- DACITE and evaluation data are publicly available [6].

II. Problem Statement

In this section, we describe a fault model for semantic bugs in DCAs, explain common sources of failures and classify them, formulate generic properties against which we check DCAs, and provide the problem statement.

A. Fault Model

A fault model includes constraints, abstractions, and actions that specify incorrect or unacceptable behavior of an engineered system [26, 71]. With respect to DCAs, a fault model describes violations of constraints on the data. Consider that $\{C_A\}$ is the set of constraints that are encoded in the application's source code, A and $\{C_D\}$ is the set of constraints that are encoded as checks in the database, D. When we write C without parentheses, we denote a single constraint from the set of constraints. Constraints contain expressions over database attributes and they are instantiated when variables that represent these attributes are assigned concrete data. We construct the following formula: $\exists d \in D, s.t.(C_A(v \mapsto d) \Rightarrow \neg C_D(v \mapsto d)) \vee (C_D(v \mapsto d) \Rightarrow \neg C_A(v \mapsto d))$. That is, if there exists an assignment of values from the data ranges that the database attributes can take to the variables in the constrains such that the negation of the instantiated constraint from the database implies an instantiated constraint from the application and vice versa, we say that there is a *semantic bug* in the DCA.

B. Categories of Semantic Bugs

We classify semantic bugs that are detected using our approach, DACITE, into the following two categories:

- *Applications-specific bugs (AB)* occur when faults are introduced in the source code of the DCAs, so that they incorrectly manipulate values of database attributes. For example, if a programmer writes code for computing insurance deductible and this computation is applied to the data object that represents a male who gave birth, then we assume that this programmer makes a semantic bug of type AB;
- *Data-specific bugs (DB)* occur when faults are introduced in the database. There are two sources for the DB bugs: incorrect explicit constraints are applied to data or some constraints are omitted. As a result, using our example for a male who gives birth, a programmer writes a conditional check in the application's source code to prevent a computation on this object while the database contains these data objects.

C. Generic Properties

When certain properties hold in DCAs, we consider them free of semantic bugs. The strongest generic property is expressed as $\forall d \in D, s.t.(C_A(v \mapsto d) \Rightarrow C_D(v \mapsto d)) \wedge (C_D(v \mapsto d) \Rightarrow C_A(v \mapsto d))$, i.e., if this theorem can be instantiated and proved for a given DCA, it is free of semantic bugs. Unfortunately, it is infeasible to extract all constraints (especially the ones that are omitted by stakeholders) and find a proof to this complex theorem. Therefore, we weaken this property to make DACITE practically applicable.

We formulate a weaker property as a situation where there exists a single assignment of data to some variables or attributes where the constraints that are infered from the application's source code are satisfiable while the corresponding constraints from the database that contain the same attributes cannot be satisfied and vice versa. That is, finding violations of this property does not mean that there are actual semantic bugs in the DCA, but DACITE generates warnings based on the violations of these constraints that enable stakeholders to localize and eventually fix these faults.

D. Overview of the Solution

In this paper, we address a three-pronged problem. First, we reverse engineer implicit constraints that are encoded in the source code of the applications. Second, we infer implicit constraints from the data that are stored in the database and that are used by the DCA. Finally, we instantiate properties based on the generic property template and determine if these properties are not violated.

The main goal of our approach is to address an important practical problem. A sound solution would guarantee that there are no bugs in the DCA if DACITE finds no violations of the properties and a complete solution would never mark some code in DCA as buggy if it is actually free of bugs. In our case, false positives and false negatives are possible, and our goal is to find semantic bugs of types AB and DB with a high degree of precision and recall. At the same time, our goal is to make this approach useful for developers who need to find semantic bugs. Although our solution is a recommendation system, which is neither sound nor complete, we aim to show a baseline of what can be achieved using only basic analyses. In section VI-A and VI-B, we show that our solution is able to detect conflicts from real DCAs and improve developers' efficiency for detecting implicit conflicts. We believe that our solution is the first tangible step to address this very difficult practical problem.

III. Illustrative example

In this section, we first declare some important notations and terms that we will use. Then, we describe how DACITE works using illustrative examples.

A. Definitions and Notations

To demonstrate how our approach works, we must first introduce some notations. In the rest of the paper, we will use `teletype` font to denote all variables, which include source code and database variables. For each function f in the subject application, let t_i denote the ith statement in f.

For each statement t_i, it has a pre-condition $t_i[Pre]$ and a post-condition $t_i[Post]$. In addition, for a (pre-/post-)condition c, $c.m$ denotes a map that links each local variable to its symbolic expression and $c.b$ denotes the branch path which is able to reach the current condition c from the beginning of the function. We use the format $\{c.m|c.b\}$ to represent the condition c. For example, $t_5[Pre]$ = {a ↦ Salary-1,000; b ↦ Age+100; c ↦ Bonus | Age>40 ∧ Sex=1} means that "The branch path to t_5 is Age>40 and Sex=1. Also, before executing t_5, a represents Salary-1,000, b represents Age+100, and c represents Bonus."

Furthermore, we annotate the program variables that are related to database attributes. We define two kinds of annotations - DBSource and DBSink. The annotation DBSource specifies that a program variable retrieves information from the database. The annotation DBSink means that a program variable updates some database cell with its value. In Figure 1, the variable age is a DBSource, which is assigned the value of attribute age after an SQL select operation in line 4. The varS is a DBSink, where its value updates the database by using SQL update operation in line 17. Since the stakeholders need to focus only on the statements which connect code with databases, the effort for annotating such variables is rather minimal [33]. Alternatively, users can also apply Meurice et al.'s approach [51] for automatically annotating the database accesses.

B. Obtaining Constraints from Source Code

Figure 1 presents some code snippet that we will use as an example. We apply static analyzer to obtain constraints from the code. The analysis process as well as the details behind the algorithm are explained in IV-B.

```
1  InitSalary(int id){
2    int age, isSenior, varS;
3    ... //read Age from DB
4    Annotation.DBSource("age", "Age");
5    ... //read Senior from DB
6    Annotation.DBSource("isSenior", "Sr_Eng");
7    if(age > 32){
8      if(isSenior == 1){
9        varS = 6500;
10       Annotation.DBSink("varS", "Salary");
11     }else{
12       varS = 5500;
13       Annotation.DBSink("varS", "Salary");}
14   }else{
15     varS = 4000;
16     Annotation.DBSink("varS", "Salary");}
17   ... //update varS to DB
18 }
```

Fig. 1. Code snippet from a sample DCA

In the beginning of the function, t_2, we map each local variable to itself and set the branch path $t_2[Pre].b$ to be $true$. Therefore, $t_2[Pre]$ is:

{age↦age; varS↦varS; isSenior↦isSenior; | $true$}

In line 4, we annotate the variable age and link it to the database attribute Age. $t_4[Post]$:

{age↦Age; varS↦varS; isSenior↦isSenior; | $true$}

In line 6, isSenior is linked to Sr_Eng. $t_6[Post]$:

{age↦Age; varS↦varS; isSenior↦Sr_Eng; | $true$}

As for the if-statement in line 7, the if-branch is taken if age>32 and else-branch is taken otherwise. Therefore, the branch path $t_8[Pre].b$ should be updated to Age>32 ∧ $true$. Similarly, $t_9[Pre].b$ will be Age>32 ∧ Sr_Eng=1 and $t_{15}[Pre].b$ will be Age≤32. Therefore, $t_9[Pre]$:

{age↦Age; varS↦varS; isSenior↦Sr_Eng; | Age> 32∧Sr_Eng=1}

Since t_9 is an assignment, varS is assigned to 6,500. Then, $t_9[Post]$ and $t_{10}[Pre]$ becomes:

{age↦↦Age; varS↦6,500; isSenior↦Sr_Eng; | Age>32∧Sr_Eng=1}

In line 10, we get the first database sink annotation, where varS has been linked to database attribute Salary. Based on $t_{10}[Pre]$, we are able to generate a constraint: Age>32 ∧ Sr_Eng=1 → Salary=6,500. By doing this over all the statements, we are able to extract three constraints from this code analysis procedure: C_1^{sc}: Age>32 ∧ Sr_Eng=1 → Salary=6,500; C_2^{sc}: Age>32 ∧ Sr_Eng≠1 → Salary=5,500; C_3^{sc}: Age≤32 → Salary=4,000. Let C^{sc} denote the set of constraints from the source code and C_j^{sc} denote the rule j in the set. We will use these constraints to find the conflicts in the final step (see III-D).

```
1  public void doGet(HttpServletRequest request,
2        HttpServletResponse response)
3    ...
4    double result = rs.getDouble(2);
5    ...
6    if(result>4){
7      p_stat = "Accept";
8      updatestat="update papers set p_stat=" +
9          p_stat +" where pID="+pid;
10   }else{
11     p_stat = "Reject";
12     updatestat="update papers set p_stat=" +
13         p_stat +" where pID="+pid;
14   }
15   ...
16 }
```

Fig. 2. Code snippet from Calculate.java in [5]

Another example from CMT DCA [5] is shown in Figure 2. Because of space limitations we show only partial code for this function, however, the full source code can be found online [5]. In line 3, the function reads the second column (Avg_rating) from table Roles and assigns the value to result. Therefore, $t_3[Post]$ is:

{result↦Avg_rating; p_stat↦p_stat; | $true$}

As for the if-statement in line 5, the if-branch is taken if result>4 and else-branch is taken otherwise:

$t_6[Pre]$: {result↦Avg_rating; p_stat↦p_stat; | Avg_rating>4}

$t_9[Pre]$: {result↦Avg_rating; p_stat↦p_stat; | Avg_rating≤4}

Since both t_6 and t_9 are assignment statements, we update the post-conditions to

$t_6[Post]$: {result↦Avg_rating; p_stat↦Accept; | Avg_rating>4}

$t_9[Post]$: {result↦Avg_rating; p_stat↦Reject; | Avg_rating≤4}

In line 7 and line 10, the database column Paper.P_stat is updated with the value of p_stat. Therefore, we are able to extract two constraints from the code analysis procedure: Roles.Avg_rating>4 → Paper.P_stat=Accept and Roles.Avg_rating≤4 → Paper.P_stat=Reject.

C. Obtaining Associative Rules from Database

A relational database follows a relational model by Codd [24] and presents information in tables with rows and columns. A table is referred to as a relation and it is a collection of objects of the same type. To obtain semantic dependencies among data items in the database automatically, we use Associative Rule Mining(ARM) algorithms. The ARM problem was first introduced by Agrawal et al. [14]. Given a table, an associative rule mining algorithm is able to generate a set of implications $X \Rightarrow Y$, where X and Y are expressions partially supported by the records in the table. The next step in DACITE is to extract constraints from the database by relying on association rule mining algorithms.

Let us assume that the database is shown in Table I. The attribute Age demonstrates the age of an employee and the attribute Sr_Eng indicates whether the person who holds

TABLE I
DATABASE TABLE FROM A SAMPLE DCA

Age	Sr_Eng	Salary	NumHouse
23	0	4,500	0
26	0	5,000	0
29	1	6,000	1
32	0	5,500	2
40	1	6,500	2
50	1	7,000	2

the record is a senior engineer or not. Salary indicates the employee's monthly salary and NumHouse shows the total number of houses that the employee possesses.

Next, we define two parameters, $supp$ and $conf$, that are used while mining associative rules. Let us assume that X and Y are attribute sets in a table and Occ(X) is defined as the occurrence of X in the table. $supp$ of rule $X \rightarrow Y$ is equal to Occ($X \cup Y$)/number of records in the table. $conf$ of rule $X \rightarrow Y$ is Occ($X \cup Y$)/Occ(X). Although the number of records in a sample database is small, results would be similar to that one with larger tables, since all ARM parameters (min $supp$, min $conf$, k) are based on ratios of total records. Thus, if we set the $supp$ to be 0.3 and $conf$ to be 0.5, the number of support records for attribute values are shown below:

Attribute value	Num. Support
{ Sr_Eng: 1 }	3
{ Sr_Eng: 0 }	3
{ NumHouse: 0 }	2
{ NumHouse: 2 }	3

Moreover, the associative rules that we can infer are:

{ Sr_Eng: 1} ⇒ {NumHouse: 2}
{ Sr_Eng: 0} ⇒ {NumHouse: 0}

However, a real estate agent may want to see relations between customer age and the number of owned houses. Thus, they could target customers who are most likely to buy a new house. In such cases, we want to see if NumHouse is related to Age or if Salary is related to Age. Since all the values in Age and Salary are distinct, the support of each value in those attributes is not high enough to generate associative rules. Our idea for solving this problem is to transfer the relational table, which has quantitative and categorical attributes, into a boolean table. The work by Srikant and Agrawal [69] refers to *Boolean Association Rules* as the problem of finding association between $true$ values in a relational table where all values are boolean values, whereas they refer to *Quantitative Association Rules* as the problem of finding association in a relational table which has quantitative and categorical attributes. By mapping the quantitative association rules problem into the boolean association rules problem, Srikant and Agrawal [69] showed that it is possible to use any boolean association rules mining algorithms (e.g. [15]) for finding quantitative association rules.

In order to map the quantitative association rules problem into the boolean association rules problem, we use the k-means clustering algorithm [36, 47] to split each quantitative attribute into k clusters (see IV-C). Then, we map the original quantitative table into the boolean value table based on the clusters, which is shown in Table II (k = 2).

TABLE II
BOOLEAN VERSION OF AN ORIGINAL DCA TABLE

Age [23, 29]	Age [32, 50]	Sr_Eng	Salary [4.5k, 5.5k]	Salary [6k, 7k]	House [0, 1]	House [2, 2]
1	0	0	1	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	1	0
0	1	0	1	0	0	1
0	1	1	0	1	0	1
0	1	1	0	1	0	1

Given the same min $supp$=0.3 and $conf$=0.5, the associative rules that we can infer now are:

R_1: {Age: [32, 50]} ⇒ {House: [2, 2]} (conf: 1.0)
R_2: {Age: [32, 50]} ∧ {Sr_Eng} ⇒ {House: [2, 2]} (conf: 1.0)
R_3: {Salary: [4,500, 5,500]} ⇒ {House: [0, 1]} (conf: 0.67)
R_4: {Age: [32, 50]} ⇒ {Salary: [6,000, 7,000]} (conf: 0.67)
R_5: {Age: [32, 50]} ∧ {Sr_Eng} ⇒ {Salary: [6,000, 7,000]} (conf: 1.0)

If we reset the min $conf$ to be 0.8, we can only infer rules 1, 2, and 5. Clearly, the number of rules that ARM can infer depends on the input parameters. We will demonstrate the influence of choosing different parameter setting on the results in the experiment evaluation section in VI-C. In the case of attributes from several tables, the approach works if we join the tables as one table.

The next step is to transform the associative rules into the logical constraints. The constraints that can be generated from the associative rules above are:

C_1^{db}: 32 ≤ Age ≤ 50 → House = 2 (conf: 1.0)
C_2^{db}: 32 ≤ Age ≤ 50 ∧ Sr_Eng = 1 → 32 ≤ House ≤ 2 (conf: 1.0)
C_3^{db}: 4,500 ≤ Salary ≤ 5,500 → 0≤ House ≤ 1 (conf: 0.67)
C_4^{db}: 32 ≤ Age ≤ 50 → 6,000 ≤ Salary ≤ 7,000 (conf: 0.67)
C_5^{db}: 32 ≤ Age ≤ 50 ∧ Sr_Eng = 1 → 6,000 ≤ Salary ≤ 7,000 (conf: 1.0)

This set of constraints will be used in the step of detecting conflicts in section III-D. Let C^{db} denote the constraint set and C_i^{db} denote the rule i in the set.

D. Detecting Conflicts

In this section, based on our example, we describe three scenarios, which include no semantic bugs, type AB semantic bug, and type DB semantic bug (see II-B).

1) No semantic bugs. We check each pair of constraints in C^{sc} and C^{db} (the details are explained in IV-D) and there is no conflict in the given example.

2) Type AB semantic bug. In Figure 1, assume a developer mistakenly puts a wrong initial number into the code in line 9, for example "varS = 5,000;", a new constraint set inferred from source code would become $C^{sc'}$: $C_1^{sc'}$: Age>32 ∧ Sr_Eng=1 → Salary=5,000; $C_2^{sc'}$: Age>32 ∧ Sr_Eng≠1 → Salary=5,500; $C_3^{sc'}$: Age≤32 → Salary=4,000.

Same as before, we check each pair of constraints in $C^{sc'}$ and C^{db}. For the constraints C_5^{db} and $C_1^{sc'}$, we have the left hand side implication as valid, which is 32≤Age≤50 ∧ Sr_Eng=1 ⇒ Age>32 ∧ Sr_Eng=1. However, the right hand side checking is unsatisfied, since UNSAT(6,000≤Salary≤7,000 ∧ Salary=5,000). We record the position of $C_1^{sc'}$ in line 9. The conflict and position will be added to the warnings list.

3) Type DB semantic bug. On the other hand, inserting or updating data in the database without restrictions may trigger other types of semantic inconsistencies in the DCA. For Table I, assume that the last two rows have been modified by another system, as the following:

Age	Sr_Eng	Salary	NumHouse
40	1	**2,500**	2
50	1	**2,000**	2

In the table, instead of inferring C_5^{db}, we would infer 32 ≤ Age ≤ 50 ∧ Sr_Eng = 1 → 2,000 ≤ Salary ≤ 2,500, which is inconsistent with the source code. Besides, it may lead to an exception being thrown, since senior developer's salary should be above that one of an entry developer. Detecting such conflicts before they manifest themselves at runtime is an essential goal of this work.

IV. OUR SOLUTION

In this section, we present core ideas behind DACITE and we describe its architecture and the workflow.

A. The Architecture of DACITE

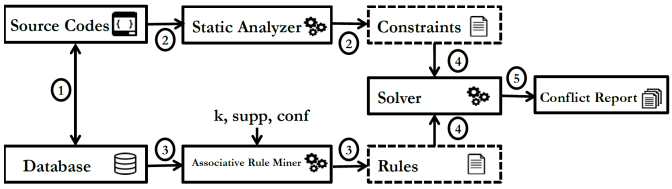


Fig. 3. DACITE's architecture.

DACITE contains three main components: a static analyzer (see IV-B), an associative rule miner (see IV-C), and a solver (see IV-D). The architecture is shown in Figure 3. The inputs of DACITE are the annotated source code and its corresponding database (1). In phase 1, the static analyzer accepts the source code and performs dataflow analysis. The output is a set of constraints in the format of $A \rightarrow B$, where both A and B are expressions (2). In phase 2, for the associative rule miner, the inputs include the database and parameter values specified by stakeholders. The outputs are associative rules inferred by the ARM algorithm. Like the constraints produced by the analyzer, the associative rules are also in the format of $A \rightarrow B$ (3). Once DACITE collects the constraints from both source code and databases, it passes the constraint pairs into a solver and checks the consistency between the source code and the database (4). Finally, DACITE records each conflict and generates the final report (5).

B. Dataflow Analysis

We perform an intra-procedural dataflow analysis to extract constraints from the source code. Our implementation is based on Soot [11]. By using control flow graphs ($CFGs$), our dataflow analysis is able to obtain information about the possible set of variable values at each point in the subject function. For each statement t_i in sequential statements $\{t_0 t_1 t_2 \dots t_n\}$ of a function, we maintain two conditions, $t_i[Pre]$ and $t_i[Post]$. Each condition maps variables to symbolic expressions along with a path condition to the condition. $t_i[Pre]$ denotes the pre-condition of t_i and $t_i[Post]$ denotes the post-condition of t_i. For a statement set T, we use $T[Post]$ to denote post-condition set of T. The classic dataflow analysis also defines transfer function TRANS for the statement t_i and meet operation JOIN_P for a statement set P, where P are predecessors for the statement t_j [40]:

$$t_i[Post] = \text{TRANS}(t_i, t_i[Pre])$$
$$t_j[Pre] = \text{JOIN}_P(P[Post])$$

In the initial condition of a function, $t_0[Pre]$, we map each local variable to itself and set the branch path to be $true$.

For the transfer function, the input includes a statement and its pre-condition. The output is the post-condition of the statement. We precisely process several basic statements in the subject language: ASSIGNMENT, BINOP, UNOP, DBSOURCE, DBSINK, etc. ASSIGNMENT, $v \leftarrow e$, assigns the expression e to the variable v. BINOP, $v = a \circ b$, computes $a \circ b$ where $\circ \in \{+, -, \times, /\}$ and assigns it to v. UNOP $v = \diamond\ a$ is an uni-operation where $\diamond \in \{++, --, -, !\}$. DBSOURCE and DBSINK ($v \leftarrow att$, $v \rightarrow att$) are DACITE annotations from a user, which link database attributes to variables.

For the meet function, we compute over-approximation. The input is a set of prior conditions and the output is the combination state for the state set. For the state condition of the output, we compute the union of all prior state conditions. We compute the over-approximation for each variable. We assign the variable to itself if it has different assignments in different branches and keep the assignment otherwise.

C. Associative Rule Mining

ARM involves four main steps in our approach.

The **first step** is to import data from the database and classify the attributes into *quantitative attributes* and *categorical attributes*. We classify numerical type attributes with a value more than ten as *quantitative attributes*. For string type attributes and numerical type attributes with number of distinct value less than ten, we classify those as *categorical attributes* (based on the size of databases, numerical type attributes that contain distinct values less than ten will have a large amount of records to support an individual value; therefore, we do not have to generate intervals.).

In the **second step,** we map the original table into a table where all the values are boolean. We have explained the reason for the table mapping in Section III-C. In our approach, we use the k-means algorithm [36, 47] to split each quantitative attribute into k intervals, where k is specified by users. We represent each quantitative value in the original table as one if the value is in the range of the splitting attribute and zero if it is not. For categorical attributes, we use each value as a new attribute in the new table. We assign one if the value is present for a given row, or zero otherwise.

In the **third step**, we apply the FP-growth algorithm [35] to generate associative rules. FP-growth is a pattern growth method for efficient mining of frequent patterns in large databases. Two parameters, $supp$ and $conf$, are provided by a stakeholder.

In the **fourth step**, we map the previous associative rules to constraints that we can use in the solver. For an interval attribute A, let $A.min$ and $A.max$ denote the lower and upper bound of the interval range ($A.min$ is equal to $A.max$ if A denotes a specific numerical value in the original table). For each associative rule $A \Rightarrow B$ where A and B are interval quantitative attributes, we map A (resp. B) to $A.min \leq A \leq A.max$ if $A.min \neq A.max$ and to $A = A.min$ (resp. B) if $A.min = A.max$. For example, assume A denotes the `age` between 40 to 50 and B denotes the `numOfCar` (the number of car) is 2, $A \Rightarrow B$ would map to constraint $40 \leq \texttt{age} \leq 50 \Rightarrow \texttt{NumOfCar} = 2$.

D. Detecting Conflicts

We use a Choco-based [4] constraint solver for detecting conflicts between the constraints from the code and the database. Let C_s (resp. C_d) denote the constraint set which is inferred from the source code (resp. database). Also, let c denote a constraint from C_s or C_d. Since all the constraints from C_s and C_d are implications (see IV-B and IV-C), c is an implication constraint and is in the format of "$A \Rightarrow B$". To explain our algorithm, we use $c.left$ to denote the left hand side of the implication c and use $c.right$ to denote the right hand side of it. Due to the limitation of the solver that we use, we only consider linear integer expressions.

We check the unsatisfiability of $c_i.right \wedge c_j.right$ only if $c_i.left \Rightarrow c_j.left$ and record the position if $c_i.left \Rightarrow c_j.left$ but $c_i.right \wedge c_j.right$ is unsatisfied. DACITE could also help the user trace back the source code location for the conflict since the location of C_s has been recorded. Thus, developers could identify the conflict location in the code based on the final report.

E. Implementation

We implemented all the components of DACITE in Java. For the static analyzer component (see IV-B), we implemented it relying on Soot-based analysis [11]. Soot is a Java optimization framework, which provides intermediate representation for static analysis. For the associative rule mining component, we implemented the technique that we presented in IV-C. Since ARM is a time-consuming algorithm, we opted to limit the size of records in the experimental database to 100K. Each component, static analysis and associative rule mining, outputs a group of constraints that are inferred respectively. JPF [8] implemented data structures for expressions that are used in symbolic execution engine. We rely on JPF's implementation for expression representation. We also implemented our own solver (see IV-D) based on Choco [4], which allows DACITE detect the inconsistencies between the source code and the databases.

V. Experimental Design

In this section, we first define three research questions (V-A). Then, we discuss subject applications (V-B) and experimental variables (V-C). Finally, we present our experimental methodologies (V-D).

We conducted several experimental studies to evaluate DACITE. i) We used DACITE to detect semantic violations in real subject applications. ii) We conducted a user case study to compare developers' time consumption and accuracy in detecting semantic bugs in the DCAs with and without using partial outputs from DACITE. The goal of the study was to evaluate if DACITE can be effectively used by developers for detecting semantic bugs. iii) We introduced more conflicts into the subjects. The goal of the study was to evaluate DACITE in a controlled setting to study how effective it is in extracting implicit constraints from the code and databases.

A. Research Questions

- **RQ**$_1$ Is DACITE able to detect conflicts from real DCAs if a database contains semantic violations?
- **RQ**$_2$ How well can DACITE improve developers' accuracy for detecting implicit conflicts (between source code and its corresponding database) as compared to the manual approach?
- **RQ**$_3$ How is DACITE's performance on detecting semantic violations by varying $conf$ and $supp$ in ARM?

In **RQ**$_1$, we examine if DACITE can extract implicit constraints from the source code and databases in open-source DCAs. By assuming existence of interoperating systems, which may have authority to manipulate the database, we insert the data in the database containing DB type semantic bugs (see II-B) and check if DACITE is able to detect these violations. Such semantic bugs are representative of those that can trigger faults in the application [30, 79, 81].

With **RQ**$_2$, we conducted a user case study to test if DACITE can help developers detect semantic violations. In order to do this, we compared developers' time consumption and accuracy in detecting semantic bugs in the DCAs with and without using DACITE. We explain our methodology behind these studies in details in V-D.

With **RQ**$_3$, we try to measure two types of metrics: conflict recall (RC) and probability of false alarms (PF). We use TP to denote *true positives*, where the conflicts reported by DACITE are indeed actual conflicts. For the FP, *false positives*, we keep track of the rules from the source code that have no conflicts with the database, but DACITE still reports them. In addition, we use FN to denote *false negative* and TN to denote *true negative*. Then, we have more specific definitions for RC and PF, where RC = TP / (TP + FN) and PF = FP / (FP + TN). RC measures the ratio of conflicts detected in the application (out of all the injected conflicts, see V-D). The higher the RC, the more conflicts are detected in the DCA. We desire higher RC and the highest possible value of RC is 100%. PF measures the ratio of alarms that are reported by DACITE and are not actual conflicts. Therefore, lower PF values are more practical to be actually useful for developers. We did not use other measures such as *accuracy* and *precision*, since the metrics are not suitable for the data where target class is rare (the target are the conflicts in our case) [50, 61]. In addition, we examine the influence of associative rule mining configuration on the results, since the rules inferred by static analysis will not change once the code is fixed, and the rules inferred from the database by associative rule mining can vary depending on different parameters. We address our claim that we are able to get a lower rate of false positives with higher support/confidence configuration for the associative rule mining. However, higher support implies lower recall for detecting conflicts, which means that we may sacrifice recall to get the implicit conflicts with higher certainty. Furthermore, We evaluate the performance of DACITE on five open-source DCAs. By evaluating the applications and databases of different sizes, our goal is to show that our technique is suitable and applicable for real-world DCAs. In Section VI, we present the time and space consumption over different stages of our technique.

TABLE III
DCA stats: size of codebases (App), size of databases (DB), number of tables (Tbl) and attributes (Attr).

DCA	App	DB	Tbl	Attr
UMAS	26.9kLOC	178KB	122	427
Potholes	11.5kLOC	1.2GB	19	87
Durbodax	14.2kLOC	25MB	28	121
Broker	16.5kLOC	1.78MB	14	46
Verse	5.3kLOC	4.27MB	15	165
A2S	2.5kLOC	n/a	n/a	n/a
BGPProgram	2.9kLOC	n/a	n/a	n/a
CMT	2.7kLOC	n/a	5	31

B. Subject Applications

We evaluate DACITE on eight open source applications that belong to different domains. Those subject applications are: 1) UMAS [12] is a course management system; 2) Potholes [10] is a movie rental system; 3) DurboDax [7] is a customer support center software; 4) Broker [3] is an application for suggesting an auto-insurance; 5) Verse [13] is a game application. 6) A2S [1] is a supermarket management system; 7) BGPProgram [2] is a flight booking system. 8) CMT [5] is a submission rating system.

Table III contains characteristics of the subject applications and their databases. The first column shows the names of the DCAs, followed by various stats. The code of the DCAs ranges from 2.5kLOC to 27kLOC. In addition, the size of the databases ranges from 178KB to 1.2GB. The total number of tables in each database ranges between 5 to 122.

C. Independent and Dependent Variables

There are two independent variables for associative rule mining, support $supp$ and confidence $conf$. We also have an independent variable k for k-means clustering. In addition, we have three independent variables for the code injection. To

answer $\mathbf{RQ}_3$, the number of injected code snippets are 18 for each application. Among those, we have nine valid snippets (no semantic bugs) for computing PF and nine snippets containing conflicts for computing RC (see V-D). Furthermore, we have three dependent variables including conflict recall (RC), probability of false alarms (PF), and execution time (ET). We experiment with DCAs by varying the independent variables for associative rule mining.

D. Methodology

To answer $\mathbf{RQ}_1$, we mock external systems to introduce type DB semantic bugs (see III-D) in the database and see if DACITE is able to catch the violations. For data generation, we use an online mock data generator, namely Mockaroo [9]. By specifying the attribute names and types, Mockaroo can generate test data including that one with advanced formulas for the attributes. The attribute relations could include mathematic operations and condition statements. Using this feature, we are able to mock data with known constraints for three systems in Table III. In addition, to simulate the real world scenario, we also randomly modified and inserted small amounts of data to introduce noise in the databases. The generated data will be used in our experiments.

To answer $\mathbf{RQ}_2$, we conducted a user study with six graduate students from authors' institution. Based on the gathered background information, the average programming experience in Java was six years and the average experience working with DCAs was four years.

To prepare the study, we selected four subject applications from Table III, which are `Durbodax`, `Broker`, `UMAS`, and `Potholes`. Based on the controlled subjects in $\mathbf{RQ}_3$, we reduced the number of both consistent and conflicting snippets to three since we had to design the study so that it can be completed in a reasonable amount of time. We recorded the snippets' locations as the ground truth and used it while analyzing the results. During the experiment, each participant had to finish four tasks in total. The four subject applications were randomly assigned to the tasks with a given sequence. Based on the sequence, we partitioned the tasks into two categories:

i) For the tasks 1 and 3 (category 1), we provided the source code and the database of a DCA to each participant. Without telling them any other information, the participants were required to locate likely semantic conflicts between the code and database. This setup simulates real scenario of how developers detect semantic bugs in practice (see II-B).

ii) For the tasks 2 and 4 (category 2), we provided not only the source code and database of a DCA, but also DACITE's output constraints for a given system. Same as before, the participants were told to locate the semantic conflicts between the source code and database. This reflects a scenario where developers need to detect semantic bugs using output constraints from DACITE.

TABLE IV
THE TASK/DCA ASSIGNMENT FOR A PAIR OF PARTICIPANTS, p AND p'. a, b, c, d ARE RANDOMLY ASSIGNED.

participant	Category #	Task #	DCA
p	category 1 (without DACITE)	task 1	a
		task 3	b
	category 2 (with DACITE)	task 2	c
		task 4	d
p'	category 1 (without DACITE)	task 1	c
		task 3	d
	category 2 (with DACITE)	task 2	a
		task 4	b

In addition, we paired every two participants as the following. For a participant p, we had a counterpart participant p' who will be assigned applications in category 1 for p as category 2, and vice versa. This is necessary to reduce possible noise due to differences in participants' skills. That is, if we have a participant who is good at finding conflicts, the effect will be noticeable in both categories. The design matrix for this study is shown in Table IV. For each task, we measured the time for each participant to finish the task and the correctness of the answers.

In order to fully answer $\mathbf{RQ}_3$, we have to have ground truth for all the violations in every single application. Since it would be extremely time-consuming to build such benchmarks manually since they do not exist in the literature [53] (and finding DCAs that contain a sizable number of native semantic bugs is rather challenging), we decided to inject some representative code snippets with and without semantic bugs into our subject DCAs. The goal was to examine if DACITE is able to detect these inconsistencies while ignoring the snippets that are consistent with the database. The injection ratio covers from 2.8% to 11.6% code statements. As we explained, we measure PF and RC based on the ground truth that we injected. For each DCA, we have nine injections which have violations and we check if DACITE is able to detect them to compute RC. We also injected nine code snippets, which are consistent with the DCA's databases. As we explained in Section V-A, we need this information in order to compute the PF and see if DACITE stays silent when it should (i.e., does not generate false alarms for consistent code statements).

We carried out the experiments using Intel Core i7-4700MQ CPU2.4GHZ with 16GB RAM. We computed RC, PF, and running time for the DCAs by varying the values of independent variables. We varied $supp$ [0.01, 0.3] with the step of 0.01 and $conf$ [0.6, 0.9] with the step of 0.1. Also, we tested different k by providing the values of 4, 10, and 40. The related results are presented in VI-C.

VI. EXPERIMENTAL RESULTS

In this section, we summarize the results of the experiments conducted to address **RQ**s in VI-A - VI-C. The threats to validity are outlined in VI-D.

A. Detecting Real Semantic Bugs (RQ_1)

We ran DACITE on three subject applications [1, 2, 5] and detected semantic bugs. Due to space limitations we provide only three examples of real semantic bugs that DACITE detected in these open-source DCAs. The goal here is to show that real-world DCAs indeed contain implicit constraints and these constraints may not be consistent with the data in the database, which may not necessarily be obvious from the database schema. Thus, it is important to rely on approaches, such as DACITE, for detecting these implicit semantic constraints or even semantic bugs. Also, the constraints derived from the source code could also be used to improve database schemas.

For the Java file Sales_Add.java in the project `A2S`, the function `updateStock()` updates the attribute `units` in the table of `product_type` to `noOfStock-noOfItem`, where `noOfStock` is the attribute `bottomlevel` in the table `product_type`. Using DACITE, we are able to obtain the constraint $true \rightarrow$ `units = bottomlevel-noOfItem`. In other words, if we run the function, the data in the database has to follow `units = bottomlevel-noOfItem` in any situation, since the left hand side of the implication is *true*.

TABLE V
RESULTS OF THE USER STUDY. * EACH TASK IS LIMITED TO 20 MINUTES.

Category	Application	$Recall_{cons}$	$Precision_{cons}$	$F-score_{cons}$	$Recall_{in}$	$Precision_{in}$	$F-score_{in}$	$Time(m)^*$
without DACITE	UMAS	0.22	0.50	0.30	0.44	0.33	0.36	20
	Durbodax	0.56	0.56	0.56	0.56	0.67	0.60	20
	Broker	0.67	1.00	0.74	0.89	0.76	0.81	15.3
	Potholes	0	0	0	0.67	0.53	0.57	20
with DACITE	UMAS	0.56	1.00	0.67	0.78	0.67	0.71	17.0
	Durbodax	0.44	0.58	0.45	0.89	0.70	0.74	9.6
	Broker	1.00	1.00	1.00	0.89	0.92	0.90	13.7
	Potholes	0.33	0.50	0.40	0.89	0.58	0.70	13.0

DACITE was able to detect the violation if the database contained inconsistent records.

For the Java file Travel.java in the project `BGPProgram`, the function `insertTravel()` updates the `FreeSeats` to `numberOfRows` × `numberOfColumns`, where `FreeSeats` is an attribute for table `Travel` and `numberOfRows`, `numberOfColumns` are coming from table `AirPlaneTypes`. In a real world scenario, the initial free seats would be the number of rows times the number of columns for a certain type of plane. If the database does not follow the rule, the system might sell more seats than the plane can take or might sell fewer seats, which would decrease profit for the airline. DACITE was able to detect the constraint and included the constraint to its report. Differently to the previous example in `A2S`, we did not put the constraint to solver since this constraint is non-linear. In such cases, we only report the constraints. As what we will show in the user study, such a report can also help developers locating the potential bugs.

For project `CMT`, we extracted two constraints from the function of `Calculate.doGet()` and reported conflicts when the database we used contains inconsistent data. The constraints include `result` $> 4 \rightarrow$ `p_stat` = "Accept" and `result` $\leq 4 \rightarrow$ `p_stat` = "Reject", where `result` is an attribute `avg_rating` for table `roles` and `p_stat` is an attribute `p_stat` for table `papers`. Although the solver cannot deal with strings, we can interpret strings as 0/1 values or numerical values. The constraints from the code indicate that the system accepts the paper if average rating is greater than 4, and rejects otherwise. A database without data-specific semantic bugs should be consistent with the code's logic.

B. Developer-based Evaluation (RQ_2)

For the user study, we are tracking four kinds of experimental results: i) logical constraints from code that a participant identifies as truly consistent/inconsistent with database (TP_{cons}/TP_{in}); ii) logical constraints from code that a participant identifies as not truly consistent/inconsistent with database (FP_{cons}/FP_{in}); iii) logical constraints from code that a participant misses as truly consistent/inconsistent with database (FN_{cons}/FN_{in}); iv) logical constraints from code that a participant misses as not truly consistent/inconsistent with database (TN_{cons}/TN_{in}). We evaluate our results using three metrics, $Recall$, $Precision$, and $F-score$. We record values of the metrics for each individual task. The definition of the metrics are as the following: $Recall_\alpha = TP_\alpha/(TP_\alpha + FN_\alpha)$, $Precision_\alpha = TP_\alpha/(TP_\alpha + FP_\alpha)$, and $F-score_\alpha = (2 \times Recall_\alpha \times Precision_\alpha) / (Recall_\alpha + Precision_\alpha)$, where $\alpha \in \{cons/in\}$. $Recall$ indicates the completeness of the results. $Precision$ indicates how accurate the result is. $F-score$ is a combined metric which considers both the $Precision$ and $Recall$ factors to measure the effectiveness. In our user study, we weight precision and recall equally [73, 74].

The average performance of each type of task in our user study is shown in Table V. In the category of "w/o DACITE", the average value of $F-Score$ for consistency (resp. inconsistency) is 0.4 (resp. 0.59). However, in the category of "using DACITE", the average value of $F-Score$ for consistency (resp. inconsistency) is 0.63 (resp. 0.76), which is greater than the previous one. The results indicate that the output information from DACITE indeed improves the accuracy of detecting semantic bugs ($\mathbf{RQ}_2$).

In terms of time consumption, the average time consumption "w/o DACITE" is 18.8 minutes while the average time consumption when "using DACITE" is 13.3 minutes. We claim that DACITE is able to reduce the time consumption for developers finding the implicit conflicts between source code and its corresponding database.

We also asked participants to fill out a survey upon completing a user study. The questions aimed at establishing the importance of semantic bugs finding and helpfulness of DACITE. These are some representative comments:

- "The db design violations arise in runtime when inputs or program states violating the constraints appear in the execution. In summary, yes, DACITE is helpful, and is very important as a tool for early bug detection";
- "The intermediate information provided by DACITE was useful, particularly the locations of the sc constraints";
- "Inconsistencies between source code and database are usually very dangerous. They should be detected and resolved as soon as possible. I think DACITE does a good job in this sense".

C. DACITE's Performance (RQ_3)

To answer the $\mathbf{RQ}_3$, we ran DACITE on several different DCAs with injected (in)consistent code snippets. The impact of the support count and confidence values on the RC and PF for the `Potholes` system (with k = 4, $conf$ = 0.6 and 0.9) is shown in Figure 4.

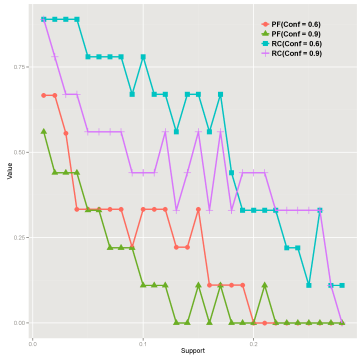
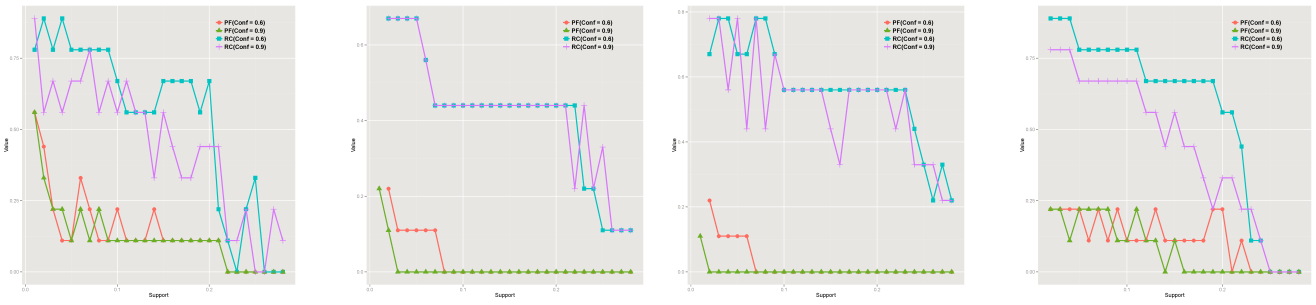


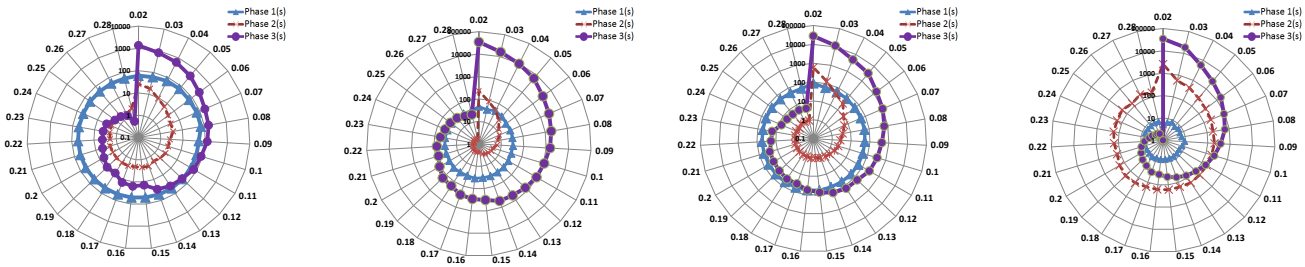
Fig. 4. Impact of support count and confidence (k=4, $conf$ = 0.6 and 0.9) on RC and PF in `Potholes`

The values for both RC and PF decrease with higher support values, which means that lower support values make DACITE infer more rule awhile using ARM and sacrifice precision.



(a) UMAS (b) Durbodax (c) Broker (d) Verse

Fig. 5. Impact of support count and confidence (k=4, $conf$ = 0.6, 0.9) on RC and PF for other DCAs



(a) UMAS (b) Durbodax (c) Broker (d) Verse

Fig. 6. Execution time (in sec) over different support values (k = 4, $conf$ = 0.6).

However, RC still remains higher than PF. In the figure, RC is around 0.5 while PF drops to 0 at $supp$ = 0.2 and $conf$ = 0.6. Comparing the confidence values of 0.6 and 0.9, we found that using higher $conf$ results in lower RC and PF.

In order to understand volatility in the results, we investigated the results closer and concluded that the main reason for these non-deterministic results was in the associative rule mining. For example, a specific rule mining procedure returned two rules for the database, r_0: 32<age<39 → NumHouse=3 and r_1: 20<age<29→ NumHouse = 1. r_0 is inconsistent with the constraint (inferred from the code snippet) c_0: 30<age<39→ NumHouse=1. However, without changing any independent variables, the associative rule mining procedure may return different rules r'_0: 27<age<39→ NumHouse=3 and r'_1: 20<age<23→ NumHouse = 1. Since the left hand side of r'_0 does not imply the left hand side of c_0, we cannot claim existence of the conflict in the code snippet.

We demonstrate the impact of configuration settings on RC and PF for four other subject DCAs in Figure 5. The downward trends are similar to Figure 4. In Figure 5(a), RC is around 0.6 at $supp$ = 0.2. However, the values drop rapidly after we increase the $supp$ greater than 0.2. Our explanation is that, for UMAS, most of the constraints from the source code are supported by one fifth of the records in the database. Generally, RC remains in high values and PF remains relatively in low values for all subject DCAs. The result shows that DACITE, as a warning system for revealing inconsistencies between code and database, could provide useful suggestions to developers.

We also recorded execution time for all analysis stages in DACITE. Phase 1 (step (2) in Figure 3) used program analysis to extract constraints from the source code. In Phase 2 (step (3) in Figure 3) associative rule mining was run against the database. Phase 3 (steps (4) and (5) in Figure 3) used the solver to find conflicts based on the constraints from the previous steps. Figure 6 shows the execution time for four of subject applications over different support values when we set the independent variable k = 4, $conf$ = 0.6. We show the running time of Potholes in the online appendix. The time values in the figures are depicted using a logarithmic scale.

The time consumption for phase 1 is stable in all the figures, since the independent variables do not influence the program analysis step. In contrast, the time consumption for phases 2 and 3 drops significantly with increased $supp$, because fewer rules were generated from databases. We observed that phase 3 consumes most of the total time when $supp$ is low and the time consumption drops dramatically if we infer fewer rules from the database. That means we can minimize the total running time by implementing a more efficient solver or improving our conflict checking strategy.

Due to the space limitation, we do not show the results of k=10 and k=40 in this paper. All the data is available in our online appendix [6]. Generally, for higher values of k, we observe decrease in running time.

D. Threats to Validity

Although DACITE is able to work on larger applications, we eliminated the tables with records greater than 100K (again, we did this only for the sake of reducing running time in our experiments; however, in reality, DACITE can be applied to larger databases). We observed the trade-off between the running time for associative rule mining algorithm and the number of useful rules that can be gleaned from the tables. For example, we have a table named "movies_info" in Potholes. The table contains millions of records (the table stores descriptions for all the movies ever created), but only has three columns: the id, the description type, and the description. In the case of such table, there are no useful rules that can be derived based on such attributes. While this does not necessarily represent a threat to validity of our results, this does show that developers should be more involved while providing an input to DACITE (e.g., by excluding meaningless

tables) to increase the number of useful rules and decrease the total running time of the approach.

The other threat to validity is that we needed to inject the code into subject DCAs in order to answer $\mathbf{RQ}_3$. The reason for relying on these injections is that the ground truth is, unfortunately, not available. Thus, we injected both consistent and inconsistent code snippets into the DCAs and then computed RC and PF metrics. In order to minimize this threat, we also examined other DCAs in public repositories($\mathbf{RQ}_1$) and detected real semantic bugs using DACITE. By doing so, we are able to locate the source code written by original developers that has interesting constraints that may be in conflict with the databases.

In this paper, the solver that we implemented is based on Presburger arithmetic. Therefore, we mainly focus on the linear integer constraints. We are currently not able to deal with constraints involving strings and arrays. Since our main contribution is in finding the implicit constraints between the source code and the database, building a richer logical solver is out of scope for this research paper. We believe that our solution is the first tangible step to address this very difficult to solve problem.

We implemented our source code analysis technique in DACITE based on intra-procedural analysis, which means that all the constraints from the source code come from individual functions. However, DACITE can be extended to also use inter-procedural analysis in the following ways: i) using annotations (we allow user annotation to link the variables with database attributes, users can annotate variables if the variable is assigned by a function which returns a database value); ii) code embedding (we can also embed function code into the function caller; we should be able to tame complexity by limiting the embedding level).

VII. Related Work

Our work is motivated by the studies of co-evolution of source codes and databases in DCAs. The work by Qiu et al. [62] presents a comprehensive co-evolution empirical study of the database schemas and source code. The paper shows that schema changes induce significant modifications in code. In our case, the changes might also be the data itself, since we focus on the implicit rules among the relations between the value of the data. The work by Maule et al. [49] uses static analysis method to identify the impact of the database schema upon object-oriented application. Dataflow analysis has been used for extracting the database interactions that DCA can make, which includes all possible insertions, updates, and stored procedure executions. They use the generated information to predict the effects for database schema changes. In our work, we focus on the variables' symbolic expressions and the holding conditions for the expression. By doing that, we extract the database attributes' relations in the code.

A different direction is to use type systems to check if database integrity is not violated [17, 18]. Contrary to using type systems, DACITE does not require programmers to adopt different type systems to express constraints; moreover, with DACITE, constraints can be re-engineered from the data using ARM. It is a subject of future work to combine refinement types with DACITE.

Other papers either focus on only database schemas or only source code aspects. Sjøberg [68] presented a technique for measuring the evolution of database schemas. Hainaut et al. [34] proposed a method to make explicit constraints present in legacy Cobol programs when migrating them to more modern data solutions such as relational databases. A related different approach has been proposed by Marcozzi et al. [48]. Meurice et al. [51] presented an approach to analyze the relations between source code and database schema. However, our work adds additional interest in its focus on database code and data inconsistencies.

Some papers [29, 37, 66] are related to the problem of protecting security between interoperating systems, where an attacker can manipulate one of the systems. One system (e.g., web application) may cause unauthorized operations on the other system (e.g., database). Huang et al. [37] describe the vulnerability and present a lattice-based static analysis approach to ensuring web application security. Fan et al. [29] introduce a new approach for conflict resolution in databases. However, the focus of DACITE is to identify semantic bugs that are introduced by interoperating systems or semantic conflicts between source code and corresponding databases.

Goeminne et al. [32] performed an empirical study on database co-evolution based on a subject application OSCAR. The authors observed that new database techniques migrate from HIB to JPA gradually while embedded SQL still remains heavily used. In addition, they claim that the majority of developers are active in both database-related and database-unrelated files. There are no specializations of developers towards only database-related activities. Their work only focuses on information from source code files, but do not explore any information from the database schemas or data in the database.

Linares-Vasquez et al. [45, 46] presented DBScribe, a novel approach for automatically generating natural language documentation at source code method level that describe database usages for a given DCA. However, their approach did not consider semantic relations between data in the database.

In summary, related papers addressed the co-evolution between source code and database. Yet, there are no existing solutions to an important problem of detecting semantic bugs. Hence, our work is entirely novel and aims at automatically revealing implicit violations between the data in databases and the source code in DCAs.

VIII. Conclusion

In this paper, we propose a novel recommendation system that enables stakeholders to automatically obtain constraints that semantically relate code to database attributes using a combination of static analysis of the source code of DCAs and associative rule mining of the databases. The main goal of our approach is to address an important practical problem: automatically detecting potential integrity violations in DCAs. We implemented the approach, namely DACITE, and evaluated it on eight open source DCAs. The results demonstrate that DACITE is effective and efficient in finding certain classes of integrity violations. We also conducted a study with developers who used DACITE for detecting semantic bugs. The results and feedback clearly emphasize that the warnings produced by DACITE are useful and enable developers to find semantic bugs faster. To the best of our knowledge, there is no previous work that focused on revealing implicit violations between the data in databases and the source code in DCAs. Thus, our technique fills an important gap in the state of research and practice of developing and maintaining DCAs.

REFERENCES

[1] "A2s. https://github.com/bruntha1991/A2S."
[2] "Bgpprogram. https://github.com/MathiasLoewe/BGPProgram."
[3] "Broker. https://github.com/fr4nkfurt/Broker."
[4] "Choco. http://www.emn.fr/z-info/choco-solver/."
[5] "Cmt. https://github.com/Sakibs/CMT_src."
[6] "Dacite online appendix. https://sites.google.com/site/daciteonlineappendix/."
[7] "Durbodax. http://se547-durbodax.svn.sourceforge.net."
[8] "Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki."
[9] "Mockaroo. https://www.mockaroo.com/."
[10] "Potholes. https://github.com/cszhangshen/Potholes.git."
[11] "Soot. http://www.sable.mcgill.ca/soot/."
[12] "Umas. https://github.com/University-Management-And-Scheduling."
[13] "Verse. https://github.com/gittekat/verse."
[14] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD'93*, pp. 207–216.
[15] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *VLDB'94*, pp. 487–499.
[16] K. Bakshi, "Considerations for big data: Architecture and approach," in *Aerospace Conference, 2012 IEEE*. IEEE, 2012, pp. 1–7.
[17] I. G. Baltopoulos, J. Borgström, and A. D. Gordon, "Maintaining database integrity with refinement types," ser. ECOOP'11, pp. 484–509.
[18] V. Benzaken and A. Doucet, "Thémis: A database programming language handling integrity constraints," *VLDB'95*, pp. 493–517.
[19] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere, "Empirical evaluation of bug linking," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 89–98.
[20] I. Bocić and T. Bultan, "Data model bugs," in *NASA Formal Methods*. Springer, 2015, pp. 393–399.
[21] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers, "Statusquo: Making familiar abstractions perform using program analysis." in *CIDR*. Citeseer, 2013.
[22] A. Cleve, "Program analysis and transformation for data-intensive system evolution," in *ICSM'10*, pp. 1–6.
[23] A. Cleve, N. Noughi, and J.-L. Hainaut, "Dynamic program analysis for database reverse engineering," in *Generative and Transformational Techniques in Software Engineering IV*. Springer, 2013, pp. 297–321.
[24] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.
[25] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems." *Commun.*, pp. 1268–1287, 1988.
[26] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models," ser. ICSE'02, pp. 241–251.
[27] A. Deutsch, R. Hull, and V. Vianu, "Automatic verification of database-centric systems," *ACM SIGMOD Record*, vol. 43, no. 3, pp. 5–17, 2014.
[28] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 151–162.
[29] W. Fan, F. Geerts, N. Tang, and W. Yu, "Conflict resolution with data currency and consistency," *Journal of Data and Information Quality (JDIQ)*, vol. 5, no. 1-2, p. 6, 2014.
[30] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 215–228.
[31] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*. ACM, 2005, pp. 213–223.
[32] M. Goeminne, A. Decan, and T. Mens, "Co-evolving code-related and database-related changes in a data-intensive software system," in *CSMR-WCRE'14*, pp. 353–357.
[33] M. Grechanik, C. Csallner, C. Fu, and Q. Xie, "Is data privacy always good for software testing?" in *ISSRE'10*, pp. 368–377.
[34] J.-L. Hainaut, A. Cleve, J. Henrard, and J.-M. Hick, "Migration of legacy information systems," in *Software Evolution*. Springer, 2008, pp. 105–138.
[35] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM SIGMOD Record*. ACM, 2000, pp. 1–12.
[36] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.
[37] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 40–52.
[38] S. Joshi and A. Lal, "Automatically finding atomic regions for fixing bugs in concurrent programs," *CoRR*, vol. abs/1403.1749, 2014.
[39] A. Katal, M. Wazid, and R. Goudar, "Big data: Issues, challenges, tools and good practices," in *Contemporary Computing (IC3), 2013 Sixth International Conference on*. IEEE, 2013, pp. 404–409.
[40] G. A. Kildall, "A unified approach to global program optimization," in *POPL'73*. ACM, 1973, pp. 194–206.
[41] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE transactions on software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.
[42] T. Kindberg and A. Fox, "System software for ubiquitous computing," *IEEE pervasive computing*, vol. 1, no. 1, pp. 70–81, 2002.
[43] C. Le Goues and W. Weimer, "Specification mining with few false positives." in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 292–306.
[44] B. Li, M. Grechanik, and D. Poshyvanyk, "Sanitizing and minimizing databases for software application test outsourcing," in *ICST'14*, pp. 233–242.
[45] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?" in *ASE'15*, 2015, pp. 36–41.
[46] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshy-

vanyk, "Documenting database usages and schema constraints in database-centric applications," in *ISSTA*. ACM, 2016, pp. 270–281.

[47] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Berkeley symposium on mathematical statistics and probability'67*.

[48] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, "A relational symbolic execution algorithm for constraint-based testing of database programs," in *SCAM*. IEEE, 2013, pp. 179–188.

[49] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *ICSE'08*, pp. 451–460.

[50] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *TSE*, vol. 33, no. 1, pp. 2–13, 2007.

[51] L. Meurice, C. Nagy, and A. Cleve, "Detecting and preventing program inconsistencies under database schema evolution," in *QRS 2016*. IEEE, 2016, pp. 262–273.

[52] ——, "Static analysis of dynamic database usage in java systems," in *International Conference on Advanced Information Systems Engineering*. Springer, 2016, pp. 491–506.

[53] L. Meurice, F. Ruiz, J. Weber, and A. Cleve, "Establishing referential integrity in legacy information systems - reality bites!" in *ICSME'14*, pp. 461–465.

[54] K. Miura, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "The impact of task granularity on co-evolution analyses," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 47.

[55] O. Mlouki, F. Khomh, and G. Antoniol, "On the detection of licenses violations in android ecosystem," in *SANER*, 2016, pp. 382–392.

[56] C. Nagy, "Static analysis of data-intensive applications," in *CSMR'13*, pp. 435–438.

[57] J. Nijjar, I. Bocić, and T. Bultan, "Data model property inference, verification, and repair for web applications," *TOSEM*, vol. 24, no. 4, p. 25, 2015.

[58] K. Pan, X. W, and T. X, "Guided test generation for database applications via synthesized database interactions," *TOSEM'14*, 2014.

[59] K. Pan, X. Wu, and T. X, "Automatic test generation for mutation testing on database applications," in *AST'13*, 2013, pp. 111–117.

[60] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *ASE'11*, pp. 73–82.

[61] F. Peters, T. Menzies, L. Gong, and H. Zhang, "Balancing privacy and utility in cross-company defect prediction," *TSE'13*, pp. 1054–1068.

[62] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *FSE'13*, pp. 125–135.

[63] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 147–157.

[64] T. Rolfsnes, L. Moonen, S. Di Alesio, R. Behjati, and D. Binkley, "Improving change recommendation using aggregated association rules," in *Proceedings of the 13th MSR*. ACM, 2016, pp. 73–84.

[65] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra, "Fault localization for data-centric programs," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 157–167.

[66] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proceedings of international conference on World Wide Web'02*. ACM, pp. 396–407.

[67] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1299–1334, 2014.

[68] D. Sjøberg, "Quantifying schema evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35–44, 1993.

[69] R. Srikant and R. Agrawal, "Mining quantitative association rules in large relational tables," in *ACM SIGMOD Record*, 1996, pp. 1–12.

[70] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *ASE'10*, pp. 289–292.

[71] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," ser. SOSP '93, pp. 203–216.

[72] C. Weiss, C. Rubio-González, and B. Liblit, "Database-backed program analysis for scalable error propagation," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 586–597.

[73] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[74] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.

[75] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proceedings of ICSME*, 2014.

[76] Y. Yang, M. Harman, J. Krinke, S. Islam, D. Binkley, Y. Zhou, and B. Xu, "An empirical study on dependence clusters for effort-aware fault-proneness prediction," in *ASE*. ACM, 2016, pp. 296–307.

[77] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Software Testing, Verification and Reliability*, vol. 22, no. 3, pp. 171–201, 2012.

[78] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie, "Software analytics in practice," *Software, IEEE*, vol. 30, no. 5, pp. 30–37, 2013.

[79] W. Zhang, C. Sun, J. Lim, S. Lu, and T. Reps, "Conmem: Detecting crash-triggering concurrency bugs through an effect-oriented approach," *TOSEM*, vol. 22, 2013.

[80] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 805–814.

[81] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *ICSE*, 2015.