A Counterexample-Guided Debugger for Non-recursive Datalog

Van-Dang Tran[1,3(✉)], Hiroyuki Kato[1,3], and Zhenjiang Hu[1,2]

[1] National Institute of Informatics, Tokyo, Japan
{dangtv,kato}@nii.ac.jp
[2] Peking University, Beijing, China
huzj@pku.edu.cn
[3] The Graduate University for Advanced Studies, SOKENDAI, Kanagawa, Japan

Abstract. The Datalog language is used in many potential applications including database queries, program analysis, bidirectional transformations, and so forth. In practice, such a Datalog program is expected to be well-written to meet requirements such as the round-tripping properties in bidirectional programming. Although verifying and debugging Datalog programs play an essential role to guarantee the expected properties of these programs, very few approaches have been proposed. The existing approaches require much users' effort in finding out unintended behaviors or unexpected computations of the Datalog program that neither counterexamples nor bug explanations are provided. In this paper, we propose an efficient approach to interactively debugging Datalog programs so that the user's burden is reduced. Specifically, we provide a syntax for users to specify properties of non-recursive Datalog programs, present a counterexample generator that verifies specified properties and generates counterexamples to show unexpected behaviors of user-written programs, and design a debugging engine combined with a dialog-based user interface to assist users in locating bugs in the programs with the generated counterexamples. We have implemented a prototype of our approach and demonstrated its feasibility and efficiency.

Keywords: Debugging · Datalog · Bidirectional transformation

1 Introduction

Datalog, a declarative logic programming language, has many applications in a variety of domains such as deductive databases [17], data integration [12], program analysis [4,11], bidirectional programming [21], and so forth. Verifying Datalog programs plays an essential role to guarantee the properties of these programs required by the applications. When a property is not satisfied, it is more important to reduce the user's burden in debugging the unexpected behavior of the program.

This kind of debugging problem, which arises when a property of a program is not satisfied, has not been well studied for Datalog. There are two challenges

B. C. d. S. Oliveira (Ed.): APLAS 2020, LNCS 12470, pp. 323–342, 2020.
https://doi.org/10.1007/978-3-030-64437-6_17

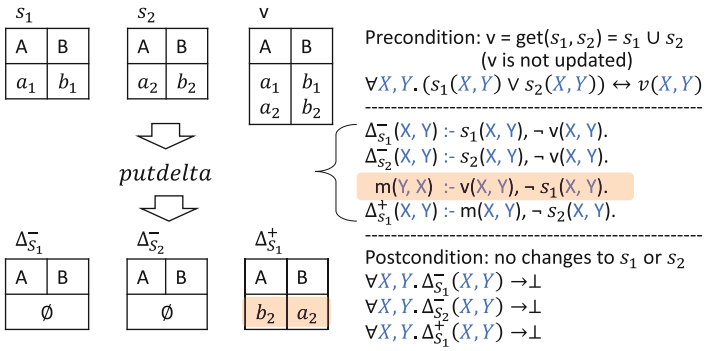

Fig. 1. Motivating example. The unexpected tuple and the buggy rule are highlighted.

in practice. The first challenge is searching for a concrete input database, i.e., a counterexample that reveals the unexpected behavior of the program. The second challenge is locating the buggy Datalog rules that break the property. By adopting the algorithmic debugging method [7], a few approaches were proposed for debugging Datalog programs [5,6,14]. However, the existing approaches neither provide users a way to specify the properties of Datalog programs nor generate counterexamples to show the incorrectness of the programs. To locate a bug, these approaches ask the users many questions about the computation correctness of the Datalog program. In other words, the users have to find out whether the Datalog program has unintended interpretations, e.g., the intention is not met by the program results. Identifying such unintended interpretations becomes costly when the input database of the program is not small.

An ideal approach to debugging would allow the user to specify the program's properties and automatically run all the checks. The properties of a program are commonly specified by a set of assertions such as equalities, domain constraints, containments, and so forth. For Datalog, which is a logic programming language in relational databases, it is intuitive for programmers to specify the assertions in the forms of relational predicates. For example, one may consider that some relations of the Datalog program must be equivalent or some relations must be empty, i.e., the corresponding predicates are always false.

We illustrate with the following example the property specifications and the debugging problem of Datalog programs.

Example 1 (Motivating Example: View Update Strategy). In this example, we consider an application of Datalog in describing view update strategies [21]. Suppose that we are given a database of two base relations $s_1(A, B)$ and $s_2(A, B)$ (Fig. 1) with a view $v(A, B)$ defined over these two relations by a union query: $v = get(s_1, s_2) = s_1 \cup s_2$. The following is a buggy Datalog program (denoted as *putdelta*) that describes a view update strategy, i.e., a description about how to update the base relations s_1 and s_2 through the view v.

$$\Delta^-_{s_1}(X,Y) \text{ :- } s_1(X,Y), \neg v(X,Y). \qquad \text{(r1)}$$

$$\Delta^-_{s_2}(X,Y) \text{ :- } s_2(X,Y), \neg v(X,Y). \quad \text{(r2)}$$
$$m(Y,X) \text{ :- } v(X,Y), \neg s_1(X,Y). \quad \text{(r3)}$$
$$\Delta^+_{s_1}(X,Y) \text{ :- } m(X,Y), \neg s_2(X,Y). \quad \text{(r4)}$$

In *putdelta*, for a relation, Δ^+ and Δ^- denote the insertion and deletion sets on the relation, respectively. Rules (r1) and (r2) state that if a tuple $\langle X, Y\rangle$ is in s_1 or s_2 but not in v, it will be deleted from s_1 or s_2, respectively. Rule (r3) checks the tuples in v but not in s_1, and stores these tuples in a mediate relation m. The last rule states that if a tuple $\langle X, Y\rangle$ is in m but not in s_2, it will be inserted into s_1. *putdelta* takes as input the states of s_1, s_2, and v to produce the delta relations of s_1 and s_2.

Such a putback program *putdelta* is required to satisfy round-tripping properties to maintain the consistency of view updates, as formulated in the existing works [10,21]. Here, we illustrate the problem with the property (called GetPut) that in the input of *putdelta*, if the view is unchanged, i.e., $v = s_1 \cup s_2$, the output of *putdelta* must be empty. We use first-order logic sentences (Fig. 1) to specify the constraints of the input (called precondition) and the constraints over the output (called postcondition).

Figure 1 shows a counterexample of GetPut that is a collection of tuples in the source tables and the view (s_1, s_2, v). Over this counterexample, the result of *putdelta* is $\Delta^-_{s_1} = \Delta^-_{s_2} = \emptyset$ and $\Delta^+_{s_1} = \{\langle b_2, a_2\rangle\}$. That means tuple $\langle b_2, a_2\rangle$ is inserted into s_1. This insertion is not expected by the postcondition. Since the input of *putdelta* satisfies the precondition but the output does not satisfy the postcondition, the GetPut property of *putdelta* is violated.

The user may wonder why tuple $\langle b_2, a_2\rangle$ of $\Delta^+_{s_1}$ occurs unexpectedly in the output of *putdelta*. From this unexpected tuple, the problem now is to detect which rules in the original Datalog program are the causes. Here, in the head of rule (r3), the variables X and Y are placed in the wrong positions and thereby some wrong tuples are derived. This bug must be fixed to make *putdelta* satisfy the GetPut property. □

We believe that for a required property of a Datalog program, the user may not only have unexpected mistakes such as typos but also have wrong intentions that do not conform to the property. Providing suggestions on how to correct the program is very useful to users but is a challenging issue. In addition, debugging is an ambiguous process that there are many possible causes for a bug. Therefore, it is essential to design an interface that lets users interact with the underlying debugging engine. For example, the user can mark suspicious rules to inspect or decide how to proceed for the bug ambiguity.

The key insight of this paper is that counterexamples play a central role in debugging Datalog programs. First, a program is buggy if and only if a counterexample exists. Second, to be useful for debugging the Datalog program, a counterexample is expected to be a realistic and simple database.

Our approach is statically generating such a counterexample rather than dynamically testing the program with randomly generated test cases as in other works such as [3]. Over the generated counterexample, bugs can be observed in the execution results of the Datalog program. Although data provenance techniques from the database literature [16] can provide useful support to explain how and why the unexpected results are derived, whether we can use this provenance information to efficiently track down the detailed source of bugs remains unclear. In this paper, we fulfill this gap by a novel method that combines the provenance information with the user interaction for resolving the ambiguity in debugging. In summary, this paper has the following contributions:

- We present a new way to use a syntactic extension of non-recursive Datalog for specifying the properties of a Datalog program.
- To explain to the user the behavior of the written Datalog program, we develop a counterexample generator that statically checks specified properties of non-recursive Datalog programs and generates counterexamples for showing why the properties are not satisfied.
- To reduce the user's effort of correcting buggy Datalog programs, we design a user interface and a provenance-based debugging engine to assist the user in locating the bugs with the counterexamples. The debugging engine provides correction hints to the user when the bugs are found.
- To demonstrate the efficiency and the usability of the proposed approach, we have implemented a prototype of the approach and evaluated it with Datalog programs in practice. The source code is available upon request.

The paper is organized as follows. Section 2 gives some background about the Datalog language with syntax extensions. In Sect. 3, we explain the design of our proposed counterexample generation method. We describe the counterexample-guided debugging approach in Sect. 4 and the experiment in Sect. 5. Section 6 presents related works. Section 7 wraps up the paper.

2 Background

A pure Datalog program is a finite set of logical rules, and each rule is an expression of the form [9]:

$$r_0(\boldsymbol{X}_0) \text{ :- } r_1(\boldsymbol{X}_1), \ldots, r_n(\boldsymbol{X}_n).$$

where $r_0, r_1, \ldots, r_n$ are relations, ":- " is a variant of the standard logical implication "←" from the rule body in the right-hand side to the rule head in the left-hand side. Each $\boldsymbol{X}_i$ ($i \in [0, n]$) is a tuple of variables. Each variable occurring in $\boldsymbol{X}_0$ must occur in at least one of $\boldsymbol{X_1}, \ldots, \boldsymbol{X_n}$ in the body.

The relations in a Datalog program are divided into two categories:

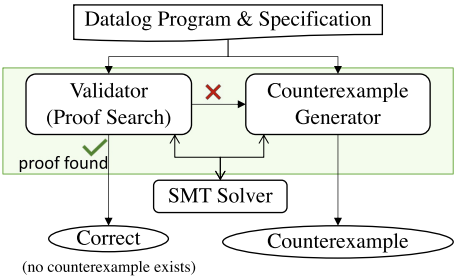

Fig. 2. Counterexample generation architecture

- *EDB relations*, which are physically stored in a relational database, called extensional database (EDB). These relations are the input of the program.
- *IDB (intensional database) relations*, which are derived from the EDB relations using the Datalog program. An IDB relation occurs in some rule heads while an EDB relation can never be in the head of a rule. An IDB relation is recursive if it appears in both the head and the body of a rule. A Datalog program is non-recursive if it has no recursive IDB relation.

We can extend Datalog by allowing negations and built-in predicates such as equality ($=$) or comparison ($<, >$) in Datalog rule bodies but in a safe way that each variable occurring in the negated atoms or the built-in predicates must also occur in some positive atoms [9]. Throughout the paper, we refer Datalog to the Datalog language with the extensions of safe negation and built-in predicates.

Let P be a Datalog program and D be the database of all the EDB and IDB relations. A tuple $\boldsymbol{A}$ in r, or a fact $r(\boldsymbol{A})$, is immediately inferred from P and D if it satisfies one of the following conditions:

- $\boldsymbol{A} \in r$, where r is an EDB relation.
- $r(\boldsymbol{A})$:- $(\neg)r_1(\boldsymbol{A}_1), \ldots, (\neg)r_n(\boldsymbol{A}_n)$. is an instantiation of a rule in P, i.e., all variables in the rule are substituted with constants. Here, a negative fact $\neg r_i(\boldsymbol{A}_i)$ holds if the fact $r_i(\boldsymbol{A}_i)$ does not hold, i.e., $\boldsymbol{A}_i$ is not a tuple of r_i in D. This is based on the Closed World Assumption (CWA) [9].

Semantically, evaluating P is computing the minimum database D such that every tuple in D is immediately inferred from D and P. In other words, we compute the least fixpoint of the immediate inference operator. In the standard bottom-up evaluation strategy for Datalog, the least fixpoint is obtained from P and the input EDB database by deriving all IDB tuples with a finite number of immediate inferences. To deal with negations in the Datalog program, the Datalog program is stratified to ensure that all the tuples of an IDB relation are derived before using any negative facts of this IDB relation in other immediate inferences. This is because if an IDB relation is incomplete, it is not sufficient to judge a negative fact of the IDB relation. The sequence of immediate inferences used for deriving a fact is called a proof of the fact and can be represented in a proof tree with different levels of the applied rules and facts.

3 Counterexample Generation

In this section, we present our approach to statically validating and generating counterexamples for a specified property of a non-recursive Datalog program.

Figure 2 shows our counterexample generation architecture. It consists of two main parts: a validator for statically checking the specified property and a counterexample generator for finding a counterexample for the property. The Datalog program with its property specification is first passed to the validator. If the validator successfully proves that the program satisfies the property, we conclude there is no counterexample. If the validator fails, the Datalog program is passed to the counterexample generator. Since many static checks such as equivalence for Datalog programs are undecidable [19], in both the validator and generator, we transform the property of the Datalog program into logical constraints that can be solved by an SMT solver, even though the termination is not guaranteed.

3.1 Specifying Program Properties

As mentioned previously, rather than introducing a new language, our approach is to use the same language to specify properties of a non-recursive Datalog program using preconditions and postconditions. By following the syntax introduced in [8,21], we allow Datalog rules to have truth constant false (denoted as $\bot$) in the head. In this way, a precondition, as well as a postcondition, is a set of Datalog rules that have the following form:

$$\bot \text{ :- } r_1(\boldsymbol{X}_1), \ldots r_n(\boldsymbol{X}_n). \qquad (*)$$

That means $\forall \boldsymbol{X}, (r_1(\boldsymbol{X}_1) \wedge \ldots \wedge r_n(\boldsymbol{X}_n)) \rightarrow \bot$, where $\boldsymbol{X}$ are all the free variables.

Example 2. Consider the GetPut property in Example 1, which says that if there is no change to the view v, there is no change to the base tables s_1 and s_2. We use non-recursive Datalog to specify the precondition as follows:

$$\begin{aligned} v^{old}(X,Y) &\text{ :- } s_1(X,Y).\\ v^{old}(X,Y) &\text{ :- } s_2(X,Y).\\ \bot &\text{ :- } v(X,Y), \neg v^{old}(X,Y).\\ \bot &\text{ :- } v^{old}(X,Y), \neg v(X,Y). \end{aligned}$$

The first two rules store the union of s_1 and s_2 in a mediate relation v^{old}, and the last two rules indicate that v is the same as v^{old}, i.e., the view does not change. And we can specify the postcondition that there is no change to the base tables as follows.

$$\begin{aligned} \bot &\text{ :- } \Delta^-_{s_1}(X,Y).\\ \bot &\text{ :- } \Delta^-_{s_2}(X,Y).\\ \bot &\text{ :- } \Delta^+_{s_1}(X,Y). \end{aligned}$$

3.2 Validation

We use an SMT solver to prove the specified property of the Datalog program by translating the property into a first-order logic (FO) sentence. If there is a proof such that the FO sentence is valid, the property is satisfied.

Our transformation from non-recursive Datalog to first-order logic is based on the standard transformation [2,9]. Let P be a non-recursive Datalog program, we inductively transform each relation r in P and the rules of the precondition and the postcondition into an equivalent FO formula φ_r as follows:

If r is an EDB relation, $\varphi_r = r(\boldsymbol{X_r}) = r(X_1, \ldots, X_{arity(r)})$.

If r is an IDB relation, i.e., r occurs in the head of m rules:

$$\begin{aligned} &r(\boldsymbol{X_r}) \text{ :- } \alpha_{1,1}, \ldots, \alpha_{1,n_1}. \\ &\ldots \\ &r(\boldsymbol{X_r}) \text{ :- } \alpha_{m,1}, \ldots, \alpha_{m,n_m}. \end{aligned}$$

The FO formula of r, if considering only the i-th rule, is $\varphi_{r,i}(\boldsymbol{X_r}) = \exists \boldsymbol{E}_i, \bigwedge_{j=1}^{n_i} \beta_{i,j}$, where $\boldsymbol{E}_i$ contains the bound variables of the i-th rule, i.e., the variables not in the rule head, and

$$\beta_{i,j} = \begin{cases} \varphi_w(\boldsymbol{Z}), \text{ if } \alpha_{i,j} \text{ is an atom } w(\boldsymbol{Z}) \\ \neg\varphi_w(\boldsymbol{Z}), \text{ if } \alpha_{i,j} \text{ is a negated atom } \neg w(\boldsymbol{Z}) \\ \alpha_{i,j}, \text{ if } \alpha_{i,j} \text{ is an equality or a built-in predicate, e.g., } x < y \end{cases}$$

By combining all the rules of r, we have:

$$\varphi_r(\boldsymbol{X_r}) = \bigvee_{i=1}^{m} \varphi_{r,i}(\boldsymbol{X_r}) = \bigvee_{i=1}^{m} \left(\exists \boldsymbol{E}_i, \bigwedge_{j=1}^{n_i} \beta_{i,j} \right)$$

By having the first-order formulas of all the IDB relations, each special Datalog rule of (*), which has $\bot$ in the head in the precondition and postcondition, is transformed into a first-order sentence: $\forall \boldsymbol{X}, (\varphi_{r_1}(\boldsymbol{X}_1) \wedge \ldots \wedge \varphi_{r_n}(\boldsymbol{X}_n)) \rightarrow \bot$. The precondition, as well as the postcondition, is a conjunction of all its FO sentences transformed from the special Datalog rules.

Let φ_{pre} and φ_{post} be the first-order sentences of the precondition and the postcondition, respectively. We employ an automated theorem prover to prove whether φ_{post} holds if φ_{pre} holds. In other words, we check whether the following first-order sentence is valid: $\varphi_{pre} \rightarrow \varphi_{post}$.

3.3 Generating Counterexamples

As mentioned previously, to assist the user in debugging a specified property, we shall generate counterexamples, which are used to guide the user to the location of bugs. The simpler the counterexamples are, the easier the user can succeed in debugging the program.

To generate a counterexample, our idea is to create a symbolic database and transform the evaluation of the Datalog program over the symbolic database with the specified property into a constraint program in Rosette [20]. The Rosette symbolic execution runtime translates the program into logical constraints that are performed by an underlying SMT solver such as Z3 [1]. The result obtained by the Rosette framework is an interpretation of the symbolic input over which the specified property of the Datalog program is violated.

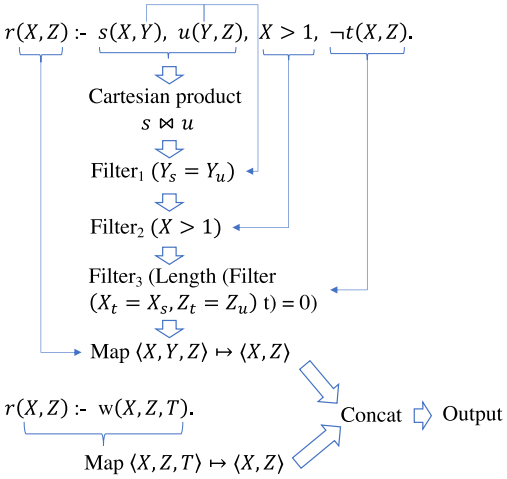

Fig. 3. Transformation from Datalog to functions

To put it more concretely, we construct a symbolic input of the source and view tables by representing each table as a list of tuples, each tuple is a list, where each element is a symbolic value. The order and the duplicates of tuples are ignored because a relation is a set of tuples rather than a list. Considering Example 1, assuming that the types of attributes `A` and `B` are integer and real, respectively, we define a symbolic table v as follows (similarly for s_1 and s_2).

```
(define-symbolic a1 integer?)  (define-symbolic a2 integer?)
(define-symbolic b1 real?)     (define-symbolic b2 real?)
(define t1 (list a1 b1))       (define t2 (list a2 b2))
(define v (list t1 t2))
```

Since string values are not supported in the underlying SMT solvers, in our transformation, we use an integer symbol for a string attribute. A value for this integer symbol will be mapped to a string value by using a predefined dictionary, where the integer value is used as an index to determine the corresponding string value. In other words, we build up a partial bijective function that maps an integer value to a string in the dictionary. Since the dictionary has finite words, we limit the values of a string attribute to be in the predefined dictionary. For example, for a relation $r(S : string)$, we define a symbolic tuple as the following:

```
(define-symbolic s1 integer?)
(assert (and (< -1  s1) (< s1 dictionary_size)))
(define t1 (list s1))
```

The assertion in the second line ensures that the value of s_1 is in the index range of the dictionary.

We evaluate a non-recursive Datalog program over a symbolic input by using four functions: Cartesian product, Filter, Map, and Concat. Figure 3 illustrates the steps for evaluating a relation r. For each rule of r, we first take a cartesian product over all positive relations in the rule body and then apply a filter (Filter_1) for the join attributes, a filter (Filter_2) for all built-in predicates, and another filter (Filter_3) for the negative relations. Over the tuples resulted from these tree filters, we use a mapping function to select the attributes appearing in the rule head[1]. If r is defined by multiple rules, we evaluate r in each rule and concatenate all the resulted tuples. For a non-recursive Datalog program, which has many IDB relations, we can inductively evaluate all the IDB relations in the program.

Example 3. For the first rule in Fig. 3, we take a cartesian product of the two positive relations s and u. The result is first filtered by Filter_2 to select only tuples, where the second attribute of s agrees with the first attribute of u, i.e., $Y_s = Y_u$. Filter_2 is applied to select the tuples satisfying $X > 1$. Filter_3 checks whether there exists a tuple $\langle X_t, Z_t \rangle$ in t that agrees with the attributes X_s and Z_u in the tuples resulted from Filter_2. The mapping function takes a projection over the three-dimension tuples and results in two-dimension tuples. Function Concat gets all the tuples computed by the two rules. □

We now turn to encode the property that is specified by the precondition and the postcondition. Recall that the precondition, as well as the postcondition, is a set of Datalog rules having constant $\perp$ in the head. To encode these Datalog rules into Rosette constraints, we first replace $\perp$ with a normal predicate, named $\emptyset_{pre}$ for the precondition and $\emptyset_{post}$ for the postcondition, and then encode the evaluation of the obtained Datalog rules into functions as presented previously. These two relations, $\emptyset_{pre}$ and $\emptyset_{post}$, are both expected to be empty. With the evaluation of $\emptyset_{pre}$ and $\emptyset_{post}$ over the symbolic input presented previously, we first encode the precondition into an assertion that the length of table $\emptyset_{pre}$ is equal to 0 as the following:

```
(assert (= 0 (length ∅pre)))
```

We then add another assertion that the length of table $\emptyset_{post}$ is greater than 0 to solve the constraint on the symbolic input that the precondition is satisfied but the postcondition is violated:

```
(solve (assert (< 0 (length ∅post))))
```

[1] It is not necessary to filter duplicates here. The duplicates will be eliminated in all the other checks and algorithms.

Algorithm 1: Counterexample generation

```
n ← 0 // The maximum size of input tables
Success ← False
while not Success do
    n ← n + 1
    foreach EDB relation r_i do // Construct a symbolic input
        Define r_i as a list of n symbolic tuples.
    // Encoding the property
    Replace ⊥ in the precondition/postcondition with ∅_pre/∅_post.
    Construct the evaluation of ∅_pre and ∅_post over the symbolic EDB relations.
    Assert the constraints for ∅_pre and ∅_post:
        (assert (= 0 (length ∅_pre)))
        (solve (assert (< 0 (length ∅_post))))
    (A list of symbol-value pairs, Success) ← Call the Rosette framework to
    resolve the constraints
    if Success then
        foreach r_i do // Instantiate all the EDB tables
            Replace each symbol with the corresponding value.
            Remove duplicates in r_i.
        return the instance of all the EDB tables.
```

Algorithm 1 summarizes the main steps in our proposed counterexample generation. Starting from 0, we increase the maximum size, denoted as n, of each input EDB table. With a value of n, we construct n symbolic tuples for each EDB table. We encode the specified property by constructing assertions corresponding to the precondition and the postcondition. We input these assertions to the Rosette framework [20] to find a value for each symbol in the input that the precondition is satisfied but the postcondition is not. If it succeeds, we stop the while loop, instantiate all the EDB symbolic tables, and eliminate duplicates. Otherwise, we continue the loop with an increased value of n.

4 Interactively Locating Bugs with Counterexamples

In this section, we present our method for interactively debugging a non-recursive Datalog program with counterexamples. Our approach consists of a user interface and an underlying debugging engine that assists the user in determining the location of bugs that cause the unexpected behavior of the program.

4.1 Checking Counterexamples

As presented in the previous section, a counterexample is an instance of the input database of the Datalog program such that the property, which is specified by the precondition and the postcondition, is not satisfied. Given an instance of the input database, to check whether the property is violated, we evaluate the output and check whether the input satisfies the precondition and the output does not satisfy the postcondition. Recall that both the precondition and the

postcondition are written in Datalog rules with a constant $\perp$ in the head. We check these conditions by replacing $\perp$ with $\emptyset_{pre}(\boldsymbol{X})/\emptyset_{post}(\boldsymbol{X})$ for the precondition/postcondition, where $\boldsymbol{X}$ are variables in the rule body, and evaluating the obtained Datalog rules. The specified property is violated if $\emptyset_{pre}$ is empty but $\emptyset_{post}$ is not empty. Any tuple appearing in $\emptyset_{post}$ is the symptom of the unexpected behavior of the Datalog program with respect to the specified property.

Example 4. Consider the *putdelta* program with an input database in Example 1 and its GetPut property specified in Example 2. To check GetPut, we check the emptiness of $\emptyset_{pre}$ and $\emptyset_{post}$ in the following rules:

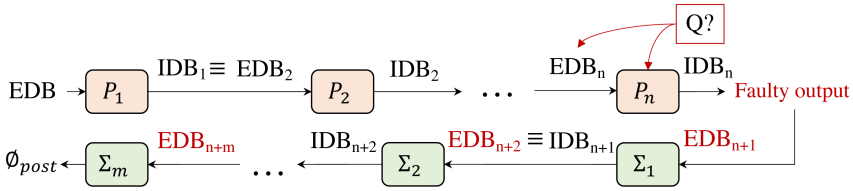

Fig. 4. Strata-based sequentialization.

$$
\begin{aligned}
v^{old}(X,Y) &\text{ :- } s_1(X,Y).\\
v^{old}(X,Y) &\text{ :- } s_2(X,Y).\\
\emptyset_{pre}(X,Y) &\text{ :- } v(X,Y), \neg v^{old}(X,Y).\\
\emptyset_{pre}(X,Y) &\text{ :- } v^{old}(X,Y), \neg v(X,Y).\\
\emptyset_{post}(X,Y) &\text{ :- } \Delta^{-}_{s_1}(X,Y).\\
\emptyset_{post}(X,Y) &\text{ :- } \Delta^{-}_{s_2}(X,Y).\\
\emptyset_{post}(X,Y) &\text{ :- } \Delta^{+}_{s_1}(X,Y).
\end{aligned}
$$

Clearly, in the result, there is no tuple in $\emptyset_{pre}$ but there is a tuple $\langle b_2, a_2\rangle$ in $\emptyset_{post}$. Therefore, GetPut is violated.

4.2 Dialog-Based User Debugging Interface

Given a counterexample, the debugging problem is to locate the buggy Datalog rules that cause the symptom that the output is faulty. It is extremely ambiguous to determine the locations of bugs since there may be many possible reasons for a fault in the output. Therefore, we allow the user to be involved in the debugging process by designing a dialog-based interface that asks the user to confirm and choose relevant options to handle the ambiguity occurring in the debugging process.

Since Datalog is a declarative programming language, the computation is not explicitly described in the Datalog program. Rather than constructing the computation tree or graph from the Datalog program as in other existing works

[5,6,14], we shall sequentialize the Datalog program to construct an order of the rules for the evaluation. In other words, we partition the original Datalog program into a sequence of smaller parts, where the final output of the program is obtained by evaluating these parts one by one in the order defined by the sequence. Similarly, we also sequentialize Datalog rules of the postcondition, where the head $\bot$ is replaced by $\emptyset_{post}$.

To construct a partition $\{P_1, P_2, \ldots, P_n\}$ of a Datalog program P, we use the well-known stratification method for Datalog [9] simplified for the case that there is no recursion in the Datalog program. Specifically, we use the precedence graph defined as the following.

Definition 1. *The precedence graph G_P of a Datalog program P is a directed graph, where nodes are the IDB relations of P and edges are relation dependencies: if $r(\boldsymbol{X})$:- $\ldots r'(\boldsymbol{Y}) \ldots$ or $r(\boldsymbol{X})$:- $\ldots \neg\, r'(\boldsymbol{Y}) \ldots$ is a rule in P, then $\langle r', r \rangle$, which represents that r' precedes r, is an edge in G_P.*

For a precedence graph, we assign to each node, which is a relation, all the rules of the relation. The rules in each node in the precedence graph form a stratum. We assign to each stratum a unique position such that if stratum P_i precedes stratum P_j in the precedence graph, then $i < j$. Clearly, each stratum in the graph can be evaluated only after all its preceding stratums are evaluated.

Figure 4 shows a program P, which is partitioned into n parts $P_1, P_2, \ldots, P_n$, and postcondition rules, which are partitioned into m parts $\Sigma_1, \ldots, \Sigma_m$. The input of P, which consists of EDB relations, is the input for the first part P_1. We evaluate the output of P by evaluating each part individually that the output of P_{i-1} (IDB_{i-1}) becomes the input of P_i (EDB_i) for every part P_i. Similarly, the output of P is the input of the postcondition rules. By evaluating $\Sigma_1, \ldots, \Sigma_m$ in this order, we obtain $\emptyset_{post}$.

Any tuple unexpectedly appearing in $\emptyset_{post}$ indicates that the specified property is violated. From this fault symptom, the debugging process is to analyze how the data is changed after each stratum to detect which stratum contains the bugs. In the input/output of a stratum, there are two types of faulty tuples: *wrong tuples*, which unexpectedly appear, and *missing tuples*, which cannot be computed as expected. For example, all the tuples in $\emptyset_{post}$ are wrong. This is caused by wrong or missing tuples in the input of Σ_m, i.e., the output of Σ_{m-1}.

For each stratum P_i, if there is a wrong/missing tuple in the output of P_i (IDB_i), we have two possible reasons: P_i contains the buggy rules; or the input of P_i, which is the output of P_{i-1}, contains wrong/missing tuples.

Since the root cause of the property violation is in the original Datalog program P, only $P_1, P_2, \ldots, P_n$ need to be inspected. Meanwhile, the stratums of the postcondition rules, $\Sigma_1, \ldots, \Sigma_m$, do not need to be inspected. They are used to detect faulty tuples in the output of P. Our underlying debugging engine automatically predicts the possible faults in the input of each stratum Σ_i. In this way, the possible faults in the output of P are detected without user interaction.

The user interaction is allowed when the underlying debugging engine inspects the stratums from P_n to P_1. At each stratum P_i, when having a faulty tuple in the output of P_i, we let the user confirm and choose one of the two

reasons for diagnosing the bugs by questioning the user about the validity of IDB_{i-1}, i.e., the input of P_i. Specifically, we evaluate all the stratums preceding P_i to obtain IDB_{i-1} and use the faulty output of P_i (IDB_i) to predict faulty tuples in IDB_{i-1}. On one hand, if the user confirms that IDB_{i-1} is valid, the underlying engine will suspect P_i to infer possible buggy rules. On the other hand, if the user finds suspiciousness in IDB_{i-1}, the underlying engine will infer possible wrong/missing tuples in IDB_{i-1} assuming P_i is correct, and then question the user to confirm the relevant faulty tuples.

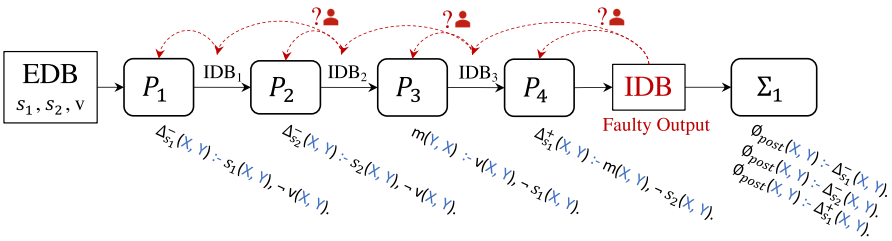

Fig. 5. Debugging interaction example.

Example 5. Figure 5 illustrates a debugging session for the *putdelta* program and its GetPut property shown in Example 1. Here, *putdelta* is stratified into four parts, P_1, P_2, P_3, P_4, corresponding to the four rules defining the four IDB relations in the program. There is only one stratum Σ_1 for the postcondition rules. □

4.3 Debugging Engine

We now present our underlying debugging engine that generates debugging details for the dialog-based user interaction and performs the debugging process based on the user's choices. Specifically, the debugging engine traverses all the stratums from the last one to the first one. At each stratum P_i, the debugging engine predicts possible faults in the input of the stratum that cause the faults observed in the output of the stratum and lets the user confirm and choose one fault. If the user confirms the input of P_i is correct, the engine suspects P_i. In contrast, if the user chooses one fault, the engine goes to the preceding stratum P_{i-1} for inspecting.

Assuming that the rules in the stratum are correct, and there is a faulty (wrong or missing) tuple in the output of the stratum, we predict faulty tuples in the input of the stratum based on the provenance information of the faulty tuple in the output that is how it is derived or how it is not derived.

For a wrong tuple in the output of the stratum, its provenance can be explained by constructing all the proof trees that are used by the stratum to derive the tuple. In our stratification strategy, each stratum contains only rules

of an IDB relation. Therefore, the maximum height of the proof trees of wrong output tuples is 1. If a wrong tuple does not belong to the IDB relation, it is derived directly from the same wrong tuple in the input of the stratum. In contrast, if a wrong tuple belongs to the IDB relation, it is derived by an immediate inference with rules in the stratum, thus its proof trees have height 1. The proof trees can be extracted from the standard bottom-up evaluation strategy [9] of Datalog by assembling all the immediate inferences.

Example 6. Considering the *putdelta* program in Example 4 and its stratification in Fig. 5, the provenance of tuple $\langle b_2, a_2 \rangle$ of $\emptyset_{post}$ in the output of the last stratum is explained by the following proof tree:

$$\frac{\Delta^+_{s_1}(b_2, a_2)}{\emptyset_{post}(b_2, a_2)} \; [\emptyset_{post}(X, Y) \text{ :- } \Delta^+_{s_1}(X, Y).]$$

where $\Delta^+_{s_1}(b_2, a_2)$ is explained by the previous stratum as the following:

$$\frac{m(b_2, a_2) \quad \neg s_2(b_2, a_2)}{\Delta^+_{s_1}(b_2, a_2)} \; [\Delta^+_{s_1}(X, Y) \text{ :- } m(X, Y), \neg s_2(X, Y).]$$

□

From the constructed proof trees, we detect all the faulty tuples in the input that must be changed to make the wrong tuples in the output disappear. For a wrong tuple, which is derived directly from the same tuple in the input of the stratum, we conclude this tuple in the input of the stratum is wrong. For a wrong tuple derived by the rules of the stratum, all the proof trees of this tuple must be deconstructed by changing the facts used in these proof trees.

Let w be the IDB relation defined in a stratum P_i, and $w(\boldsymbol{A}_0)$ be a wrong tuple in the output of P_i. A proof tree of $w(\boldsymbol{A}_0)$ has the following form:

$$\frac{(\neg)r_1(\boldsymbol{A}_1) \quad \ldots \quad (\neg)r_n(\boldsymbol{A}_n)}{w(\boldsymbol{A}_0)} \; [w(\boldsymbol{X}_0) \text{ :- } (\neg)r_1(\boldsymbol{X}_1), \ldots, (\neg)r_n(\boldsymbol{X}_n).]$$

Here, we apply the rule $w(\boldsymbol{X}_0)$:- $(\neg)r_1(\boldsymbol{X}_1), \ldots, (\neg)r_n(\boldsymbol{X}_n)$ with the facts $(\neg)r_1(\boldsymbol{A}_1), \ldots, (\neg)r_n(\boldsymbol{A}_n)$ to infer $w(\boldsymbol{A}_0)$. Since $w(\boldsymbol{A}_0)$ is derived if all the facts $(\neg)r_1(\boldsymbol{A}_1)$, ..., and $(\neg)r_n(\boldsymbol{A}_n)$ hold, changing one of $(\neg)r_1(\boldsymbol{A}_1), \ldots, (\neg)r_n(\boldsymbol{A}_n)$ is sufficient to make $w(\boldsymbol{A}_0)$ not derived, and thus correct $w(\boldsymbol{A}_0)$. In other words, $w(\boldsymbol{A}_0)$ is wrong because one of the facts $(\neg)r_1(\boldsymbol{A}_1), \ldots, (\neg)r_n(\boldsymbol{A}_n)$ is wrong. We exclude facts that are from EDB relations because the EDB database is not computed by the Datalog program. We raise a question to the user interface to let the user confirm and choose one wrong tuple. This is repeatedly performed for each proof tree of each wrong tuple in the output of P_i.

Remark 1. A fact $\neg r(\boldsymbol{A})$ is wrong iff $r(\boldsymbol{A})$ is missing. This follows from the closed world assumption (CWA).

A missing tuple, which is not derived in the output of a stratum, is explained by any proof tree that fails to be constructed. The failed proof tree cannot be completed because of some facts that are required but do not hold. As presented previously, in our stratification strategy, each stratum contains only rules of an IDB relation that the proof trees of a tuple have maximum height 1. A proof tree, which has height 1, is constructed by instantiating a rule in the stratum. To avoid constructing an infinite number of proof trees that are not related to the context of the Datalog program, as other approaches [16], we restrict the Datalog program to its active domain, which is the set of all constants appearing in the EDB relations and the program. Specifically, only values in the active domain are used to instantiate a rule. In this way, we obtain a finite number of proof trees for a tuple in the output.

We detect the faulty tuples in the input that cause a missing tuple in the output as follows. If the missing tuple does not belong to the IDB relation defined by the rules in the stratum, we conclude it is missing in the input of the stratum. In contrast, we construct a proof tree of the missing tuple by instantiating a rule in the stratum and then find all the facts not holding in the rule body. Clearly, these faulty facts explain the missing tuple in the output of the stratum. In this way, by constructing all the proof trees, we enumerate all possible faults in the input and raise a question to the user for choosing the most suitable fault. To reduce the number of possible faults, we also prefer the smaller faults to the bigger ones. A fault is smaller if the number of faulty facts in the fault is smaller. The smaller a fault is, the more easily it can be fixed.

We have predicted all the faults (wrong and missing tuples) in the input of a stratum based on the assumption that the rules in the stratum are correct. At the user interface level, we have raised questions to the user to confirm the faults in the input that cause the faulty tuples in the output. Since a stratum contains only rules of an IDB relation, named r_i, changing the rules in the stratum can only correct the faulty tuples of r_i in the output. Therefore, for the faulty tuples of r_i, if in the input, there is no possible fault or the user confirms no predicted fault is suitable, we can conclude that the rules in the stratum contain the bugs and start inspecting the stratum's rules.

Given a faulty tuple in the output of a stratum and assuming that all the tuples in the input are correct, the problem is to determine which rules of r_i are wrong or whether a rule is missing. For a wrong tuple in the output, to locate the corresponding buggy rules, we use the wrong tuple's proof trees constructed before. Specifically, all the rules applied in these proof trees are wrong since they must be changed to make the wrong tuple disappear in the output. For a missing tuple in the output, the user has two ways to fix the rules for producing the missing tuple. The first option is changing one of the rules in the stratum so that it can produce the missing tuple. The second option is adding to the stratum a new rule that can be applied to derive the missing tuple.

To assist the user in correcting the buggy rules in the stratum, we give the user correction hints by showing the proof trees of the faulty tuples and showing the input and the output expected for adding/changing the rules. To be

efficient, at each stratum, we show all these observations to the users for finding the cheapest way to correct all the bugs found.

Example 7. We illustrate our debugging approach by considering the *putdelta* program in Example 1 with another property, called PutGet [21], specified as follows. There is no rule for the precondition, and the postcondition is:

$$s_1^{new}(X,Y) \text{ :- } s_1(X,Y), \neg\Delta_{s_1}^-(X,Y) \quad \text{(r5)}$$
$$s_1^{new}(X,Y) \text{ :- } \Delta_{s_1}^+(X,Y). \quad \text{(r6)}$$
$$s_2^{new}(X,Y) \text{ :- } s_2(X,Y), \neg\Delta_{s_2}^-(X,Y). \quad \text{(r7)}$$
$$v^{new}(X,Y) \text{ :- } s_1^{new}(X,Y). \quad \text{(r8)}$$
$$v^{new}(X,Y) \text{ :- } s_2^{new}(X,Y). \quad \text{(r9)}$$
$$\bot \text{ :- } v^{new}(X,Y), \neg v(X,Y). \quad \text{(r10)}$$
$$\bot \text{ :- } v(X,Y), \neg v^{new}(X,Y). \quad \text{(r11)}$$

That means if we apply delta relations, $\Delta_{s_1/s_2}^{\pm}$ obtained from the *putdelta* program, to the source relations, s_1 and s_2, and calculate the view v^{new} again, we expect v^{new} to be the same as the initial view v. Let us consider a counterexample of PutGet as the following: $s_1 = \{\langle a_1, b_1\rangle\}$, $s_2 = \emptyset$, $v = \{\langle a_1, b_1\rangle, \langle a_2, b_2\rangle\}$. Over this counterexample, the result of *putdelta* is: $\Delta_{s_1}^- = \Delta_{s_1}^- = \emptyset$, $\Delta_{s_1}^+ = \{\langle b_2, a_2\rangle\}$. Thus, $v^{new} = \{\langle a_1, b_1\rangle, \langle b_2, a_2\rangle\}$, leading to that $\emptyset_{post} = \{\langle a_2, b_2\rangle, \langle b_2, a_2\rangle\}$ in the rules (r10) and (r11). Therefore, the PutGet property is violated.

Figure 6 illustrates how the causes of the wrong tuples $\emptyset_{post}(a_2, b_2)$ and $\emptyset_{post}(b_2, a_2)$ are predicted. Here, the *putdelta* program is stratified into P_1, P_2, P_3, P_4 and the PutGet precondition is stratified into Σ_1, Σ_2, Σ_3, Σ_4.

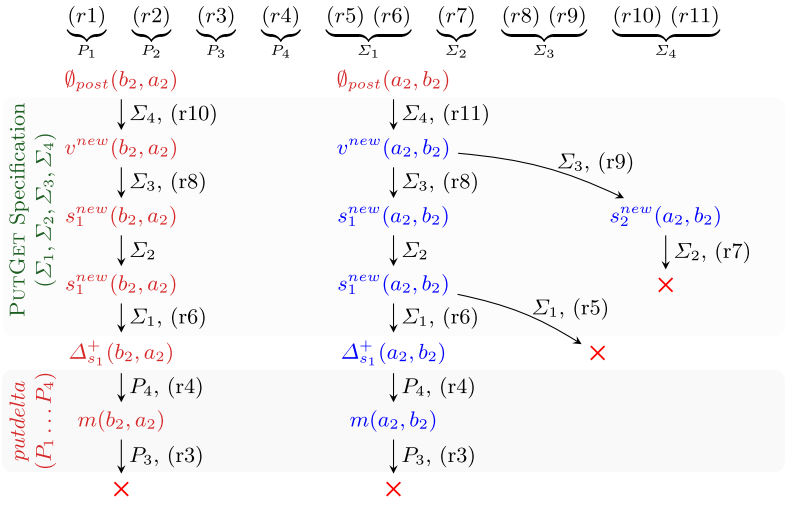

Fig. 6. Debugging demonstration.

For the wrong tuple $\emptyset_{post}(b_2, a_2)$, by using its proof trees at each stratum of $\Sigma_1, \Sigma_2, \Sigma_3$ and Σ_4, we have wrong tuples $v^{new}(b_2, a_2)$, $s_1^{new}(b_2, a_2)$, $s_1^{new}(b_2, a_2)$, and $\Delta_{s_1}^+(b_2, a_2)$, respectively. Since stratum Σ_2 does not contain any rules defining s_1^{new}, the wrong tuple $s_1^{new}(b_2, a_2)$ in the output of Σ_2 is simply derived from this wrong tuple $s_1^{new}(b_2, a_2)$ in the input of Σ_2.

For the wrong tuple $\emptyset_{post}(a_2, b_2)$, at stratum Σ_4, we predict a wrong fact $\neg v^{new}(a_2, b_2)$ in the input of Σ_4. That means $v^{new}(a_2, b_2)$ is missing. At stratum Σ_3, there are two possible proof trees corresponding to rules (r8) and (r9), respectively. Therefore, there are two possible causes of $v^{new}(a_2, b_2)$: $s_1^{new}(a_2, b_2)$ is missing or $s_2^{new}(a_2, b_2)$ is missing. We continue to predict the causes of each of these tuples $s_1^{new}(a_2, b_2)$ and $s_2^{new}(a_2, b_2)$. Eventually, some predicted causes are invalid. For example, at Σ_2, the cause of the missing tuple $s_2^{new}(a_2, b_2)$ is a missing tuple $s_2(a_2, b_2)$ which cannot be fixed because s_2 is an EDB relation. There is only one valid cause: $\Delta_{s_1}^+(a_2, b_2)$ is missing.

Table 1. Debugging results. ✓ indicates that the property is satisfied.

ID	Program	Rules (program & properties)	Counterexample generation time (s)	Counterexample size (tuples)			Number of questions
				DeltaDis	GetPut	PutGet	
1	luxuryitems	12	8.721	✓	✓	2	0
2	ukaz_lok	13	7.162	✓	✓	2	0
3	message	21	10.652	3	2	3	1
4	poi_view	23	10.08	✓	2	3	1
5	all_cars	24	11.116	3	2	3	2
6	newpc	26	10.294	✓	✓	3	1
7	products	28	13.614	✓	✓	4	1
8	purchaseview	29	9.153	✓	5	✓	0
9	vehicle_view	30	Timeout	–	–	–	–
10	koncerty	32	47.951	✓	✓	5	2
11	phonelist	33	11.035	4	3	4	1

After predicting the faults in the output of P_4, i.e., the output of the *putdelta* program, the user interaction is triggered. At stratum P_4, assuming P_4 is correct, the cause of the wrong tuple $\Delta_{s_1}^+(b_2, a_2)$ is a wrong tuple $m(b_2, a_2)$ and the cause of the missing tuple $\Delta_{s_1}^+(a_2, b_2)$ is a missing tuple $m(a_2, b_2)$. Here, a question of confirming whether $m(b_2, a_2)$ is wrong and whether $m(a_2, b_2)$ is missing is raised to the user interface. If the user confirms there is no faulty tuple, the debugging engine will inspect P_4; in contrast, it goes to stratum P_3. For inspecting P_4, since there is only one rule (r4) that is used in the proof tree of $\Delta_{s_1}^+(b_2, a_2)$ and $\Delta_{s_1}^+(a_2, b_2)$, (r4) is a buggy rule. For P_3, because no fault in the input of P_3 is predicted, the engine inspects P_3 without user interaction. Interestingly, both the choices of inspecting P_4 or going to P_3 can detect the bug that can be solved. Specifically, changing $m(X, Y)$ in (r4) to $m(Y, X)$ can make $\Delta_{s_1}^+(b_2, a_2)$ disappear and make $\Delta_{s_1}^+(a_2, b_2)$ appear in the output, and thus PutGet satisfied. Similarly, changing $m(Y, X)$ in (r3) to $m(X, Y)$ can also correct the program. □

5 Implementation and Experiment

We have implemented a prototype for our debugging approach in Ocaml and integrated it with Rosette [20] and Z3 [1] as the SMT solvers for our counterexample generation. The user can interact with our system via a command-line tool. By the tool, the user can start a debugging session with a counterexample which is automatically generated by the tool or given by the user.

To evaluate our approach, we use non-recursive Datalog programs collected in [21]. These programs are written for implementing practical view update strategies that are required to be well-defined (called the DeltaDis property) and satisfy the round-tripping properties, i.e., GetPut and PutGet, with the corresponding view definitions to guarantee the consistency between the views and the source tables. We randomly add bugs to these programs and run an experiment to evaluate the performance of our approach in debugging these programs. Specifically, we measure the time for generating counterexamples, the size of the generated counterexamples, and the number of questions used to ask the user for locating the bugs. The experiment is performed on a computer of 2 CPUs and 4 GB RAM running Ubuntu Server LTS 16.04. We set up a timeout of 1 min for generating counterexamples.

Table 1 summarizes the results of our experiment. The time for generating counterexamples and the size of counterexamples almost increase against the number of rules in the program and the specified properties. The generating time also depends on the difficulty of the bugs and the complexity of Datalog rules. For example, `phonelist` has a smaller generating time than `koncerty` because the rules of `phonelist` are more straightforward. `products` has a bigger generating time than `purchaseview` because PutGet is usually more complex than GetPut. For `vehicle_view`, the counterexample generator does not terminate after the maximum allowed running time. The results show that the number of questions used in locating bugs is usually small. This number depends on the complexity of the program and the difficulty of the bugs. Some simple programs such as `luxuryitems` have no question, meanwhile, some bigger programs such as `all_cars` and `koncerty`, which contain more bugs or more user-written rules, need more questions with the user interaction to find the buggy rules.

6 Related Work

Algorithmic debugging [18], also known as declarative debugging, is a semi-automatic debugging technique that is based on the answers of the programmer to a series of questions generated automatically by the algorithmic debugger. Due to its abstraction level, this technique is relevant to declarative programming languages such as Datalog. Some approaches [5,6,14] have been proposed to apply algorithmic debugging to Datalog. These existing approaches can assist the user after a fault (i.e., a counterexample) is detected but suffer from the well-known scalability problems of algorithmic debugging [7] that more user interaction is required in the debugging process. In our approach, we strengthen

the algorithmic debugging technique applied to non-recursive Datalog by statically generating minimum-size counterexamples for the debugging process. We exploit provenance techniques [13,15,16] to automatically predict the root causes of the observed faults of the Datalog programs for reducing the human effort of answering the questions raised by the algorithmic debugger.

7 Conclusion

In this paper, we have presented a novel debugging approach to non-recursive Datalog programs. Our framework assists users in checking and generating counterexamples for the programs with properties prespecified by users and then uses counterexamples to guide the users to the location of bugs via a dialog-based interface. The experimental results show the performance of our approach.

Acknowledgments. We would like to thank Meng Wang and the anonymous reviewers for their insightful comments on this paper. This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099.

References

1. Z3: Theorem prover (2018). https://z3prover.github.io
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995)
3. Amaral, C., Florido, M., Santos Costa, V.: PrologCheck – property-based testing in prolog. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 1–17. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_1
4. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA, pp. 243–262 (2009)
5. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A theoretical framework for the declarative debugging of Datalog programs. In: Semantics in Data and Knowledge Bases, pp. 143–159 (2008)
6. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A new proposal for debugging Datalog programs. In: WFLP 2007 (2007)
7. Caballero, R., Riesco, A., Silva, J.: A survey of algorithmic debugging. ACM Comput. Surv. **50**(4), 60:1–60:35 (2017)
8. Calì, A., Gottlob, G., Lukasiewicz, T.: Datalog±: a unified approach to ontologies and integrity constraints. In: ICDT, pp. 14–30 (2009)
9. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). TKDE **1**(1), 146–166 (1989)
10. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Theory and Practice of Model Transformations, pp. 260–283 (2009)
11. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416 (2012)
12. Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update exchange with mappings and provenance. In: VLDB, pp. 675–686 (2007)

13. Herschel, M., Hernández, M.A.: Explaining missing answers to SPJUA queries. PVLDB **3**(1), 185–196 (2010)
14. Köhler, S., Ludäscher, B., Smaragdakis, Y.: Declarative Datalog debugging for mere mortals. In: Datalog in Academia and Industry, pp. 111–122 (2012)
15. Köhler, S., Ludäscher, B., Zinn, D.: First-order provenance games. In: In Search of Elegance in the Theory and Practice of Computation, pp. 382–399 (2013)
16. Lee, S., Köhler, S., Ludäscher, B., Glavic, B.: A SQL-middleware unifying why and why-not provenance for first-order queries. In: ICDE, pp. 485–496 (2017)
17. Sáenz-Pérez, F., Caballero, R., García-Ruiz, Y.: A deductive database with Datalog and SQL query languages. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 66–73. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25318-8_8
18. Shapiro, E.Y.: Algorithmic program diagnosis. In: POPL, pp. 299–308 (1982)
19. Shmueli, O.: Equivalence of Datalog queries is undecidable. J. Logic Program. **15**(3), 231–241 (1993)
20. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: PLDI, pp. 530–541 (2014)
21. Tran, V.D., Kato, H., Hu, Z.: Programmable view update strategies on relations. PVLDB **13**(5), 726–739 (2020)