Dependency-Aware Metamorphic Testing of Datalog Engines

Muhammad Numair Mansur
numair@mpi-sws.org
MPI-SWS
Germany

Valentin Wüstholz
valentin.wustholz@consensys.net
ConsenSys
Austria

Maria Christakis
maria.christakis@tuwien.ac.at
TU Wien
Austria

ABSTRACT

Datalog is a declarative query language with wide applicability, especially in program analysis. Queries are evaluated by Datalog engines, which are complex and thus prone to returning incorrect results. Such bugs, called query bugs, may compromise the soundness of upstream program analyzers, having potentially detrimental consequences in safety-critical settings.

To address this issue, we develop a metamorphic testing approach for detecting query bugs in Datalog engines. In comparison to existing work, our approach is based on rich precedence information capturing dependencies among relations in the program. This enables much more general and effective metamorphic transformations. We implement our approach in DLSmith, which detected 16 previously unknown query bugs in four Datalog engines.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Datalog, metamorphic testing, fuzzing

ACM Reference Format:
Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598052

1 INTRODUCTION

Datalog [20] is a declarative query language, which, at its core, is based on a decidable fragment of first-order logic. Due to its simplicity and expressiveness, a wide range of applications have been implemented using Datalog, including static analyzers for Java [6, 38], a parallelizing compiler framework [16], a binary disassembler [18], and security checkers for smart contracts [7, 19, 51], to name a few.

Queries are evaluated by *Datalog engines*, such as Soufflé [24], bddbddb [53], and DDlog [44]. Such engines are complex, especially since they typically employ advanced query transformation, optimization, and compilation techniques to improve their performance and scalability. As a result of this complexity, Datalog engines are prone to *query bugs* [34]. A query bug causes the engine to return incorrect results that, for example, contain more, fewer, or different entries than they should. These bugs are severe—they may compromise the soundness of an upstream program analyzer, leading to catastrophic consequences in safety-critical settings.

It is, therefore, critical to develop automatic validation techniques for detecting query bugs in Datalog engines. Finding such bugs, however, is impossible without an oracle, that is, a specification of the *expected* results. Differential testing [36] is a test generation technique that would overcome the oracle problem by running multiple Datalog engines on the same input programs and looking for disagreement in the results. Nevertheless, there is no unified syntax for Datalog, and each engine understands a (very) different dialect; for instance, Soufflé enables large-scale, logic-oriented programming, whereas Formulog [4] provides support for constructing and reasoning about SMT formulas.

Metamorphic testing [15] is another test generation technique for addressing the oracle problem. In our context, metamorphic testing would transform a Datalog program such that the result of the new program has a known relationship with the result of the original program. For example, the new result could be equivalent to the original result, it could be contained in the original result, or it could contain the original result.

In fact, the first metamorphic testing technique for Datalog engines was recently implemented in a tool called queryFuzz [34]. However, this technique is limited in the metamorphic transformations it can perform. In particular, it selects an existing rule in a given Datalog program and carefully modifies it without considering the surrounding program. More specifically, its transformations only consist in adding an atom to a rule, removing an atom from a rule, or modifying a rule variable. queryFuzz requires that such changes do not introduce negation and may only be performed for a specific set of rules (i.e., those at the highest stratum) such that the resulting program is still valid. In short, transformations in queryFuzz are limited to ones that can be performed locally without considering the entire program.

This paper overcomes these limitations by inferring an *annotated precedence graph* for a given Datalog program, capturing rich information about any dependencies among program relations. Hence, this graph provides a global view of the program, thereby allowing for more radical transformations, including adding entirely new rules, removing existing rules, and handling negation. At the same time, our approach incorporates all existing queryFuzz transformations. In other words, we significantly extend the range of possible transformations, and thus, increase the effectiveness of metamorphic testing in finding query bugs in Datalog engines. Moreover, by defining all our transformations on the annotated precedence graph, our approach can easily support many Datalog dialects.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598052

We implemented this approach in our tool DLSmith, which we used to test six Datalog engines, each supporting a different dialect. DLSmith detected 16 previously unknown query bugs in four of these engines. All bugs were confirmed and eleven were fixed; only two could have been found by queryFuzz. An engine developer commented: *"The bugs found are hidden deep inside the query plan generation and optimization pipeline. Due to the complexity of Datalog rules and data for triggering the error, the bugs are very hard to find with regular unit testing. Automatic tools are extremely helpful to identify the issue and improve the robustness of our Datalog engine."*

Contributions. Our paper makes the following contributions:

(1) We present the most comprehensive and effective metamorphic testing approach for detecting query bugs in Datalog engines to date.
(2) We implemented our approach in the publicly available tool DLSmith[1].
(3) We evaluated DLSmith on six Datalog engines supporting different dialects; our tool detected 16 previously unknown query bugs in four of these engines and received very positive feedback from their developers.

Outline. The next section provides necessary background on Datalog. Sect. 3 gives an overview of our approach, while Sects. 4 and 5 explain the technical details of its key components. In Sect. 6, we describe the implementation of DLSmith. We present our experimental evaluation in Sect. 7, discuss related work in Sect. 8, and conclude in Sect. 9.

2 BACKGROUND

In this section, we give an overview of Datalog programs and their associated precedence graphs.

2.1 Datalog Programs

Rules. A *term* is either a variable $x, y, z, \ldots$ or a constant $a, b, c, \ldots$. An *atom* is an expression of the form $R(\vec{U})$, where R is a *relation symbol* of arity m and $\vec{U}$ is an m-vector of terms, e.g., $M(x, y, a)$. A *ground atom* is an atom without variables, e.g., $M(a_1, \ldots, a_m)$, where a_i are constants. A *Datalog rule* is an expression of the form

$$R(\vec{U}) \leftarrow R_1(\vec{U_1}), \ldots, R_n(\vec{U_n}).$$

where $R_i(\vec{U_i})$ for $1 \leq i \leq n$ are atoms. Note that atoms can refer to the same relation. The expression to the left of $\leftarrow$ is the *head* of the rule, and the expression to the right is the *body*. Any variable appearing in $\vec{U}$ must also appear in some $\vec{U_i}$. A relation R can have more than one rule, each of which is identified by a unique *rule number* k, where k ranges between 1 and the total number of rules for the relation.

Programs. Relation symbols are divided in two categories. First, there are *input relations* whose contents are given in the form of *facts* (ground atoms). These are commonly referred to as *extensional database (EDB) relations*. We use F to denote the set of facts. Second, there are *intensional database (IDB) relations* that are defined by Datalog rules, and one of them is specified as *output*. A Datalog program P is a finite set of facts and Datalog rules. P is *recursive* if a relation symbol appears in both the head and the body of a rule.

[1] https://github.com/Rigorous-Software-Engineering/dlsmith

For example, the following is a recursive Datalog program with three facts and two rules:

```
// facts (representing edges)
E(1,2). E(2,3). E(3,4).
// rules (computing the transitive closure of E)
C(x,z) :- E(x,z).
C(x,z) :- C(x,y), C(y,z).
```

The input to the program is `E` (EDB relation), and the output, `C` (IDB relation), represents the transitive closure of the edge relation `E`.

Stratified Datalog. A *stratification* of a Datalog program assigns a non-negative integer, called a *stratification number* or *stratum*, to every IDB relation in the program such that, for every rule, the following hold:

- For every positive (i.e., not negated) atom R_i in the rule body, the stratum of R_i is greater than or equal to the stratum of rule head R.
- For every negative (i.e., negated) atom R_i in the rule body, the stratum of R_i is strictly greater than the stratum of rule head R.

Stratification allows providing well defined semantics for evaluating Datalog programs [30], and consequently, most Datalog engines only support stratifiable programs. The evaluation of a program starts with the highest stratum, for which a fixpoint is computed. The computed results of any IDB relation in the highest stratum are then used in the second highest stratum. The process is repeated until all strata are traversed.

P is a *stratified Datalog program* if it does not contain recursion involving negation. For example, the following program is not stratifiable because relation `P` negatively depends on relation `L`, which again depends on `P`:

```
L(a) :- M(a), P().
P(a) :- R(a), not L(a).
```

2.2 Precedence Graphs

A Datalog program P has an associated directed graph, called *precedence graph* and denoted by G_P. G_P has a node for each relation in the program and an edge from node N to M whenever a relation N is in the body of a certain rule and relation M is the head of the same rule. A precedence graph is, therefore, used to capture dependencies between relations in the program. If a relation appears positively in the rule body, the corresponding edge is annotated with label +, otherwise with −. When P is non-recursive, its precedence graph is by definition acyclic. As an example, consider the program in Fig. 1 with its associated precedence graph.

Definition 2.1 (Precedence Graph). Given a Datalog program P, a precedence graph $G_P = (V, E, \theta, \lambda)$ is a directed, labeled hypergraph, where V is a set of nodes. Each node in V represents a unique relation in P. Function $\theta : Q \rightarrow V$ assigns a relation in Q to a node in V, where Q is the set of all relations in P. $E \subseteq (V \times V)$ is a set of directed edges. Function $\lambda : E \rightarrow sign$, where $sign$ is $\{+, -\}$, assigns labels to edges.

3 OVERVIEW

In this section, we give an overview of our approach (shown in Fig. 2), which is divided into four phases. On a high level, it uses a

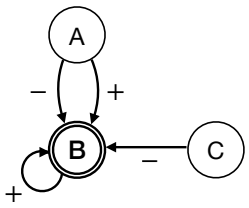
```
// declarations
A(X: number, Y:number).
B(X:number, Y:number).
C(X:number, Y:number, Z:number).

.output B

// facts
A(1,2).
A(2,3).
A(3,4).

//rules
B(x,y) :- B(x,y), !A(y,x).
B(x,y) :- A(x,y), !C(x,y,y).
```

(a)



(b)

Figure 1: A simple Datalog program (a) and its associated precedence graph (b).

seed program to generate a random annotated precedence graph, applies metamorphic transformations on this graph, and compares the results of the programs corresponding to these two graphs.

The *first* phase takes as input a seed Datalog program S and produces a random precedence graph G_P. It does so by first extracting all relation symbols in S. For each relation symbol, it generates a graph node, called a *seed node*. Next, it randomly generates a number of new nodes, called *generated nodes*. G_P is then produced using these seed and generated nodes. Note, however, that no incoming edges are added to seed nodes, that is, the corresponding rules remain unchanged—this allows handling complex seed programs containing unique language features (e.g., SMT formulas in Formulog rules). Since program S in Fig. 2 consists of only one relation, G_P contains one seed node (`fib`); node `a` is randomly generated.

The *second* phase takes S and G_P as input and produces a corresponding program P as well as its result O_P. To achieve this, the graph annotator first uses G_P to generate an *annotated precedence graph* $\overline{G_P}$, which extends G_P by decorating its nodes and edges with properties. S and $\overline{G_P}$ are then used by the program generator to produce P, which is in turn executed to compute O_P. Note that the construction of $\overline{G_P}$ by the graph annotator ensures that P is stratifiable and passes all syntactic, semantic, and type checks of the target Datalog engine.

Under the hood, the program generator starts by creating a new relation for each node in $\overline{G_P}$. Relations created from seed nodes are called *seed relations*, and all others are *generated relations*. Rules for a seed relation are copied directly from S (since the corresponding seed node in $\overline{G_P}$ has no incoming edges). Rules for a generated relation are created based on the incoming edges of the corresponding node. Edge properties in $\overline{G_P}$ (*number*, *sign*, and *vars*) are directly reflected in program *syntax*. For example, when considering the edge from `fib` to `a`, its properties denote that atom `fib(x,x)` (*vars*) appears positively (*sign*) in the first rule (*number*) for relation `a`. Node properties (*stratum* and *ancestry*) are *semantic*; they are computed using a lightweight static analysis on $\overline{G_P}$.

The *third* phase takes S and $\overline{G_P}$ as input and produces a new program P_{tr}, which constitutes a metamorphic transformation of P, as well as its result $O_{P_{tr}}$. In particular, we transform P to obtain P_{tr} such that $O_{P_{tr}}$ has a known relation with O_P. Examples of such relations are *equivalent*, *contracting*, and *expanding* transformations, i.e., $O_P \equiv O_{P_{EQU}}$, $O_P \supseteq O_{P_{CON}}$, and $O_P \subseteq O_{P_{EXP}}$. This is achieved with the graph transformer, which applies graph rewrite rules on $\overline{G_P}$ to obtain $\overline{G_{P_{tr}}}$, while again ensuring that no incoming edges are added to seed nodes. Next, the program generator, which is the same as in the previous phase, converts $\overline{G_{P_{tr}}}$ into transformed program P_{tr}. In the end, P_{tr} is executed to compute $O_{P_{tr}}$. In Fig. 2, the graph transformer adds two edges from `fib` to `a` to generate a metamorphically equivalent annotated precedence graph $\overline{G_{P_{EQU}}}$.

The *fourth* phase compares results O_P and $O_{P_{tr}}$ according to oracle *tr*. A bug report is generated when the oracle does not hold.

The following two sections explain the key components of our approach in more detail, namely, the graph annotator and the graph transformer.

4 GRAPH ANNOTATOR

Recall that, in the second phase of our approach, the graph annotator takes as input a randomly generated precedence graph, G_P, and produces an annotated precedence graph, $\overline{G_P}$, by decorating the nodes and edges in G_P with *property-value pairs*. Such graphs are also known as *property graphs*.

Fig. 3 shows an annotated precedence graph for generating the program of Fig. 1. As in a standard precedence graph, we represent each relation symbol in the program by a node, called *relational node*. Each relational node maintains two properties: *stratum* and *ancestry*. The output relation in the program, which is a relational node in the graph, is additionally called an *output node*—the output node is shown with a double line in the figure. We call edges between relational nodes *relational edges*. Each relational edge maintains three properties: *number*, *sign*, and *vars*. We represent a ground atom (i.e., fact) by a distinct type of node, called *fact node*—fact nodes are shown with dotted lines in the figure. We associate each fact node with a relational node using a *fact edge*. We denote fact values as labels in fact nodes, e.g., label "`1,2`" for fact `A(1,2)`. Fact nodes and fact edges are property-less.

Definition 4.1 (Annotated Precedence Graph). Given a Datalog program P, an annotated precedence graph, denoted by $\overline{G_P} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, is a directed, attributed hyper-graph. V is the set of relational nodes, V_F the set of fact nodes, and $O \in V$ the output node. $E \subseteq (V \times V)$ is the set of relational edges and $E_F \subseteq (V_F \times V)$ the set of fact edges. There is a fact edge from *every* node $u \in V_F$ to some node $v \in V$. Function $\theta : Q \rightarrow V$ assigns a relation in Q to a relational node in V, where Q is the set of all relations in P. Similarly, $\theta_F : F \rightarrow V_F$ assigns a fact in F to a fact node in V_F, where F is the set of all facts in P. Relational nodes are assigned properties using function $\lambda : V \times K^v \rightarrow vals^v$, where $K^v = \{stratum, ancestry\}$ is the set of node property keys and $vals^v$ the set of node property values such that $stratum \in \mathbb{N}$ and $ancestry \in \{+, -, ?, \text{none}\}$. For output node O, λ is defined to assign $stratum = 0$ and $ancestry = +$. Relational edges are assigned properties using function $\mu : E \times K^e \rightarrow vals^e$, where $K^e = \{number, sign, vars\}$ is the set of edge property keys and $vals^e$ the set of edge property values such that $number \in \mathbb{N}$, $sign \in \{+, -\}$, and *vars* is a tuple of variables and/or constants.

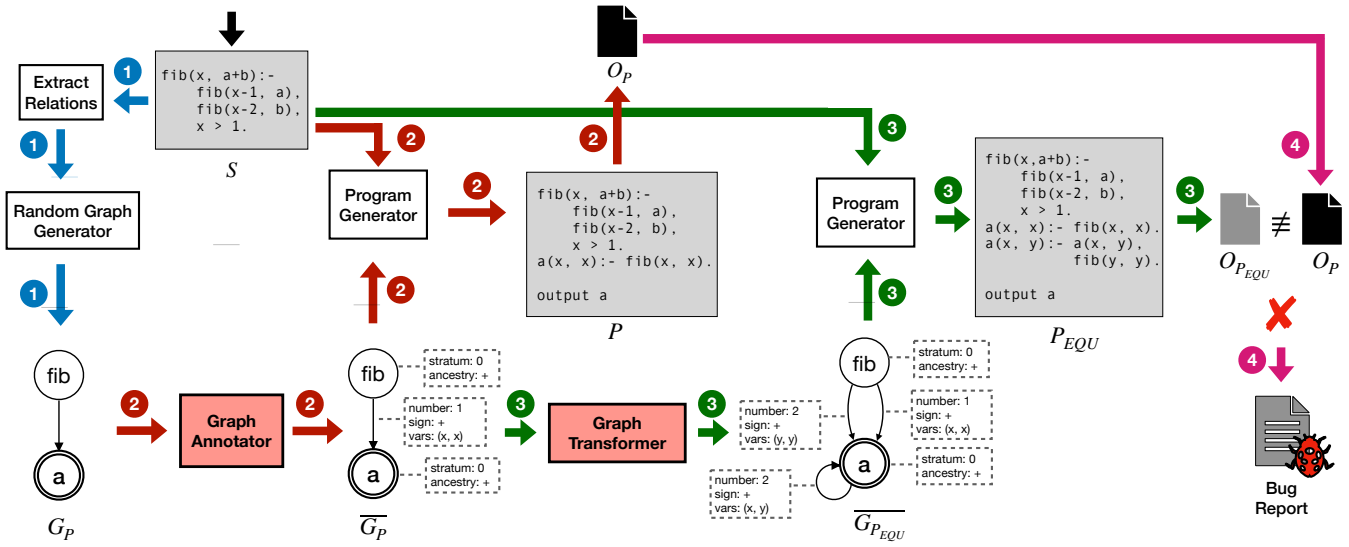


Figure 2: Overview of our approach.

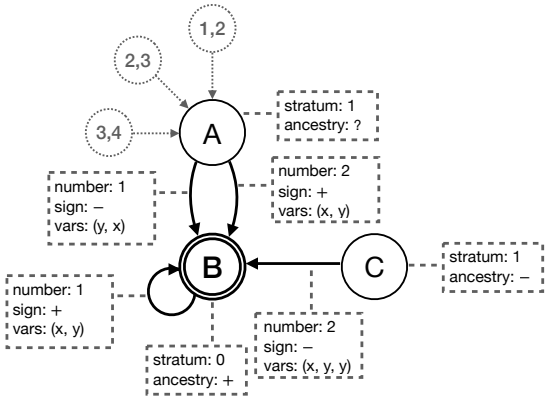


Figure 3: Annotated precedence graph for generating the program of Fig. 1.

On a high level, extending a standard precedence graph to an annotated one involves three steps: (1) randomly adding fact nodes and edges, (2) randomly generating properties for relational edges, and (3) inferring properties for relational nodes.

Fact nodes and edges. A fact node represents a ground atom of a relation. A fact edge e_f from a fact node F to a relational node N denotes that F represents a ground atom of relation N. In Fig. 3, we have three fact nodes connected to relational node A; these represent the three facts in the program of Fig. 1.

Relational-edge properties. As in standard precedence graphs, there is a relational edge e from a relational node N to a relational node M if N appears in the body of a rule r and M is the head of the same rule. Property *number* for e is k, where k is the rule number for r. *sign* is + if N appears positively in r, otherwise it is $-$. *vars* is $\vec{U}_i$ if N is the ith atom in the body of r. Note that these properties are syntactic; they are later used by the program generator to produce a valid Datalog program.

In Fig. 3, we have an edge from B to B with property values *number* = 1, *sign* = +, and *vars* = (x, y). In the program of Fig. 1, we therefore have a recursive relation B, where B appears positively in the first rule with variables (x, y). In Fig. 3, we also have an edge from C to B with property values *number* = 2, *sign* = $-$, and *vars* = (x, y, y). As a result, in Fig. 1, relation C appears negatively in the second rule for B with variables (x, y, y).

We call an edge e *positive* if property *sign* = +, otherwise we call it *negative*. We call a path between two relational nodes in $\overline{G}$ a *dataflow path* (denoted by π). π from N to M is *positive* if the number of negative edges in π is even, otherwise π is *negative*. It is important to note that data flows monotonically along any dataflow path between N and M. In the case of a positive path, an increase or decrease in data in N increases or decreases (although not necessarily strictly) the data in M, respectively. In the case of a negative path, an increase or decrease in data in N decreases or increases (although not necessarily strictly) the data in M, respectively.

Relational-node properties. Property *ancestry* for a node N is + if all (possibly infinite) dataflow paths from N to output node O are positive; *ancestry* for N is $-$ if all (possibly infinite) paths from N to O are negative; *ancestry* is ? if there is at least one positive and one negative path from N to O; and *ancestry* is none if there is no dataflow path from N to O. We say that a node is in *positive ancestry* of O if *ancestry* is + for that node, the node is in *negative ancestry* if *ancestry* is $-$, the node is in *unknown ancestry* if *ancestry* is ?, and the node is *not in the ancestry* if *ancestry* is none. We compute the value of *ancestry* for a node N by performing a backward depth-first traversal of $\overline{G}$ from O to N.

Property *stratum* for N is the stratification number of N. Essentially, it is the largest number of negative edges along any path from N to O in $\overline{G}$. Note that, in a stratified Datalog program, all relations have a finite stratum, that is, the precedence graph of a stratified

program has no cycle that contains a negative edge. For example, in Fig. 3, C has *stratum* = 1 and *ancestry* = − since there is a negative path from C to B with one negative edge. A has *ancestry* = ? since there is at least one positive and one negative path from A to B.

Relational-node properties are semantic; they are computed by the graph annotator with a lightweight static analysis of $\overline{G}$ and are later used by the graph transformer to apply valid metamorphic transformations.

5 GRAPH TRANSFORMER

In this section, we define several primitive rewrite rules for annotated precedence graphs, introduce a methodology for specifying metamorphic transformations using these rules, and provide concrete example transformations.

5.1 Graph Rewrite Rules

Graph rewriting transforms a host graph, in our context the annotated precedence graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, by adding, removing, or altering the properties of graph elements (nodes or edges) using declaratively defined rules. A *graph rewrite rule* $\mathcal{R}(**g, **atr)$ is a variadic function, i.e., a function of indefinite arity, that takes as input a number of host graph elements (represented as $**g$) and rewrite attributes (represented as $**atr$); it returns a result graph $\overline{G_{tr}}$.

Add rewrite rules. Add rewrite rules transform $\overline{G}$ by adding a relational node, a fact node, or a relational edge between two existing relational nodes.

AddRelNode. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule adds a relational node n with property values *nvals* to $\overline{G}$; we denote it as $\mathcal{R}_{\text{AddRelNode}}(**g, **atr)$, where $**g = \{n\}$ and $**atr = \{nvals\}$. The result graph is $\overline{G_{tr}} = (V', V_F, O, E, E_F, \theta', \theta_F, \lambda', \mu)$, where $V' = V \cup \{n\}$ and θ', λ' only differ from θ, λ by including n.

AddFact. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule adds a fact node f and a fact edge e_f from f to relational node $v \in V$; we denote it as $\mathcal{R}_{\text{AddFact}}(**g, **atr)$, where $**g = \{f, v\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V, V'_F, O, E, E'_F, \theta, \theta'_F, \lambda, \mu)$, where $V'_F = V_F \cup f$, $E'_F = E_F \cup e_f$, and θ'_F only differs from θ_F by including f.

AddRelEdge. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule adds a relational edge e_r with property value *evals* from u to v, where $u, v \in V$; we denote it as $\mathcal{R}_{\text{AddRelEdge}}(**g, **atr)$, where $**g = \{u, v\}$ and $**atr = \{evals\}$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E', E_F, \theta, \theta_F, \lambda, \mu')$, where $E' = E \cup e_r$ and μ' only differs from μ by including e_r.

Del rewrite rules. Del rewrite rules transform $\overline{G}$ by deleting a relational node, a fact node, or a relational edge between two existing relational nodes.

DelRelNode. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule deletes a node $v \in V$ along with its edges; we denote it as $\mathcal{R}_{\text{DelRelNode}}(**g, **atr)$, where $**g = \{v\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V', V_F, O, E', E_F, \theta', \theta_F, \lambda', \mu')$, where $V' = V \setminus v$, $E' = E \setminus E_v$ if E_v is the set of incoming and outgoing edges of v, and θ', λ' only differ from θ, λ by excluding v.

DelFact. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule deletes a fact node f associated with relational node $v \in V$ and the fact edge e_f from f to v; we denote it as $\mathcal{R}_{\text{DelFact}}(**g, **atr)$, where $**g = \{f, v\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V, V'_F, O, E, E'_F, \theta, \theta'_F, \lambda, \mu)$, where $V'_F = V_F \setminus f$, $E'_F = E_F \setminus e_f$, and θ'_F only differs from θ_F by excluding f.

DelRelEdge. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule deletes a relational edge $e \in E$; we denote the rule as $\mathcal{R}_{\text{DelRelEdge}}(**g, **atr)$, where $**g = \{e\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E', E_F, \theta, \theta_F, \lambda, \mu')$, where $E' = E \setminus e$ and μ' only differs from μ by excluding e.

Mod rewrite rules. Mod rewrite rules transform $\overline{G}$ by modifying property values of an existing relational node or edge.

ModRelNode. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule modifies the property values of a node $v \in V$; we denote it as $\mathcal{R}_{\text{ModRelNode}}(**g, **atr)$, where $**g = \{v\}$ and $**atr = \{nvals\}$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda', \mu)$, where λ' only differs from λ by assigning property values *nvals* to v.

ModRelEdge. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule modifies the property values of an edge $e \in E$; we denote it as $\mathcal{R}_{\text{ModRelEdge}}(**g, **atr)$, where $**g = \{e\}$ and $**atr = \{evals\}$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu')$, where μ' only differs from μ by assigning property values *evals* to e.

5.2 Specifying Metamorphic Transformations

Recall that the graph transformer (see Fig. 2) applies graph rewrite rules on the annotated precedence graph $\overline{G_P}$ of program P to obtain $\overline{G_{P_{tr}}}$, which will then be converted into transformed program P_{tr}. Here, *tr* is the metamorphic relation that holds between the output of P (O_P) and that of P_{tr} ($O_{P_{tr}}$). Specifically, we have

- $O_P \equiv O_{P_{EQU}}$ for equivalent transformations,
- $O_P \supseteq O_{P_{CON}}$ for contracting transformations, and
- $O_P \subseteq O_{P_{EXP}}$ for expanding transformations.

We enforce relation *tr* by applying a graph rewrite rule $\mathcal{R}$ on a set of graph elements that satisfy a precondition ϕ. In general, we define a metamorphic transformation as a triple

```
assume(ϕ(**g))

**atr = Ḡ_P.generate_attributes(**arg)
Ḡ_{P_tr} = Ḡ_P.R(**g, **atr)

assert(out(Ḡ_P) ≈_tr out(Ḡ_{P_tr}))
```

stating that if precondition ϕ holds for graph elements $**g$, then applying rewrite rule $\mathcal{R}$ on $**g$ establishes a relation *tr* between the output of P (`out`($\overline{G_P}$)) and that of P_{tr} (`out`($\overline{G_{P_{tr}}}$)). In this context, designing a metamorphic transformation essentially consists in defining precondition ϕ, one or more rewrite rules, and an attribute generation scheme implemented in method `generate_attributes`.

As an example, consider rewrite rule $\mathcal{R}_{\text{AddRelEdge}}(u, v, evals)$, which adds an edge e from relational node u to relational node v. In the transformed program P_{tr}, this means that a new atom is added in a rule for R, where $R = \theta^{-1}(v)$ is the relation corresponding to v. When $u \in V$, where V is the set of all relational nodes in $\overline{G_P}$, and $v \in V_{\text{none}}$, where V_{none} is the set of all nodes that are not in the ancestry of the output node O, we have an equivalent (EQU) transformation. This is because there is no dataflow path from v to

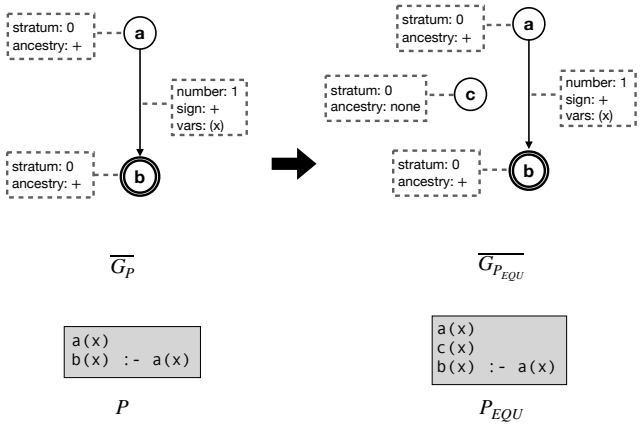


Figure 4: An example EQU-AddRelNode transformation.

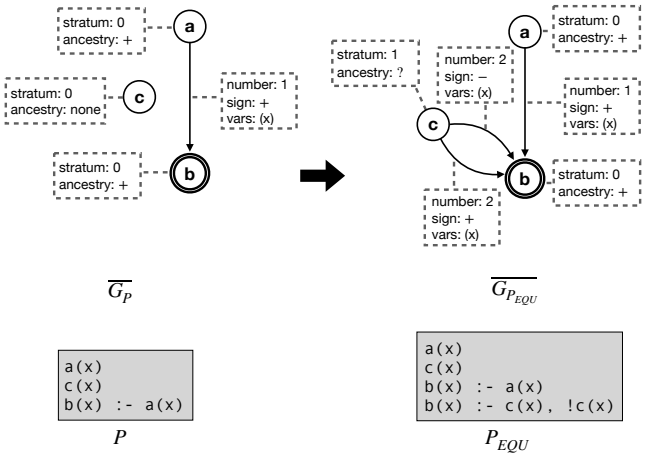


Figure 5: An example EQU-AddRelEdges transformation.

O, and thus, when adding e from u to v, there is still no data flowing to O through e. We represent this transformation as follows.

`assume(`$u \in \overline{G_P}$`.get_all_nodes() ⋀`
$v \in \overline{G_P}$`.get_nodes_with_ancestry(none))`

$evals = \overline{G_P}$`.generate_attributes(`u, v`)`
$\overline{G_{P_{EQU}}} = \overline{G_P}.\mathcal{R}_{\text{AddRelEdge}}(u, v, evals)$

`assert(out(`$\overline{G_P}$`) ≡ out(`$\overline{G_{P_{EQU}}}$`))`

Method `get_all_nodes` retrieves all nodes in the annotated precedence graph, whereas method `get_nodes_with_ancestry` retrieves all nodes with a particular ancestry, in this case `none`. Method `generate_attributes` returns values for properties *number*, *sign*, and *vars* of new edge e.

5.3 Example Metamorphic Transformations

We now present a sample of the transformations implemented in DLSmith—these are the transformations that detected query bugs in the Datalog engines we tested. For the remaining transformations in DLSmith, see Tab. 1.

EQU-AddRelNode. This is an equivalent transformation adding a new relational node (see Fig. 4 for an example).

`assume(`*true*`)`

$nvals = \{stratum : 0,\ ancestry : \text{none}\}$
$\overline{G_{P_{EQU}}} = \overline{G_P}.\mathcal{R}_{\text{AddRelNode}}(n, nvals)$

`assert(out(`$\overline{G_P}$`) ≡ out(`$\overline{G_{P_{EQU}}}$`))`

Here, $\{stratum : 0,\ ancestry : \text{none}\}$ is the implementation of `generate_attributes`. Recall that property *stratum* of a node is the largest number of negative edges along any path from the node to the output in the annotated precedence graph. Here, since *ancestry* is `none`, there is no path from the new node to the output, and thus, *stratum* must be 0.

EQU-AddRelEdges. This is an equivalent transformation adding a positive and a negative relational edge between two existing nodes (see Fig. 5 for an example). In particular, it adds two edges e and e' from relational node u to relational node v in $\overline{G_P}$, where $sign = +$ for e and $sign = -$ for e'. Node v may be any relational node in $\overline{G_P}$, but to ensure that the resulting program P_{EQU} is stratifiable, node u may not be a descendant of v.

`assume(`$v \in \overline{G_P}$`.get_all_nodes() ⋀`
$u \in \overline{G_P}$`.get_all_nodes() \ `$\overline{G_P}$`.get_descendants(`v`))`

$k = \overline{G_P}$`.get_max_rule_number(`v`) + 1`
$args = \overline{G_P}$`.generate_vars(`v, k`)`
$evals = \{number : k,\ sign : +,\ vars : args\}$
$evals' = \{number : k,\ sign : -,\ vars : args\}$
$s = \overline{G_P}$`.get_stratum(`v`) + 1`
$a =$ **`if`** $\overline{G_P}$`.get_ancestry(`v`) == none` **`then`** `none` **`else`** `?`
$nvals = \{stratum : s,\ ancestry : a\}$
$\overline{G_{P_{EQU}}} = \overline{G_P}.\mathcal{R}_{\text{AddRelEdge}}(u, v, evals)$
$\overline{G_{P_{EQU}}} = \overline{G_{P_{EQU}}}.\mathcal{R}_{\text{AddRelEdge}}(u, v, evals')$
$\overline{G_{P_{EQU}}} = \overline{G_{P_{EQU}}}.\mathcal{R}_{\text{ModRelNode}}(u, nvals)$

`assert(out(`$\overline{G_P}$`) ≡ out(`$\overline{G_{P_{EQU}}}$`))`

Note that adding two such edges to any node makes the corresponding rule compute an empty result. Therefore, for the transformation to be equivalent, we apply it on a new rule of an existing relation R, where $R = \theta^{-1}(v)$. Method `get_max_rule_number` retrieves the total number of existing rules for R, and thus, k must be `get_max_rule_number(`v`) + 1`. Method `generate_vars` generates random variables, which however satisfy the type constraints of R. Finally, we update the properties of node u to reflect the added edges—*stratum* is incremented by 1 due to the negative edge, and if there is a path from v to the output, then *ancestry* becomes ? due to the edges having different *sign* values.

EQU-AddSelfEdge. This transformation adds a positive self edge e to an existing relational node v, that is, the edge connects v

Table 1: Remaining metamorphic transformations implemented in DLSmith (grouped by oracles EQU, CON, and EXP).

Transformation	Description
EQU-AddFact	Adds a fact node to a relational node that is not in the ancestry of the output
EQU-DelFact	Deletes a fact node from a relational node that is not in the ancestry of the output
EQU-DelRelNode	Deletes a relational node that is not in the ancestry of the output
EQU-DelRelEdge	Deletes an incoming relational edge from a node that is not in the ancestry of the output
CON-AddFact	Adds a fact node to a relational node that is in negative ancestry of the output
CON-DelFact	Deletes a fact node from a relational node that is in positive ancestry of the output
CON-DelRelEdges	Deletes all incoming relational edges from a node that is in positive ancestry of the output
EXP-AddFact	Adds a fact node to a relational node that is in positive ancestry of the output
EXP-DelFact	Deletes a fact node from a relational node that is in negative ancestry of the output
EXP-DelRelEdge	Deletes an incoming relational edge from a node that is in positive ancestry of the output

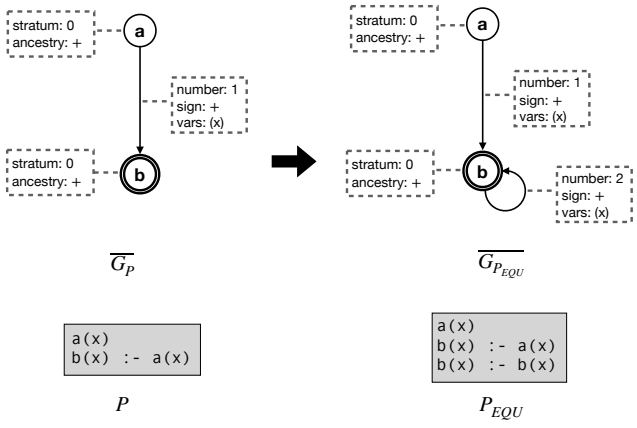


Figure 6: An example EQU-AddSelfEdge transformation.

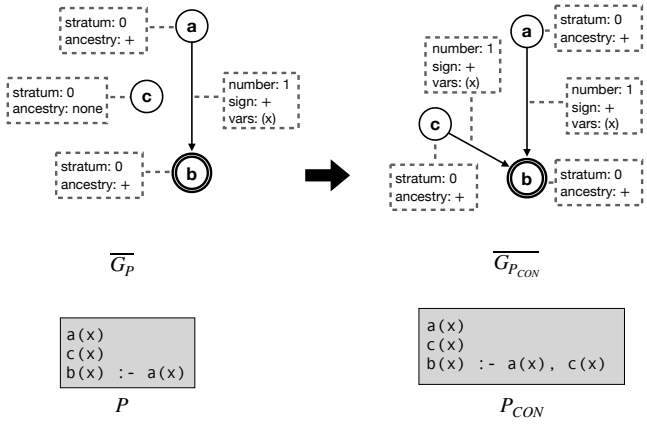


Figure 7: An example CON-AddRelEdge transformation.

to itself (see Fig. 6 for an example). Similarly to EQU-AddRelEdges, for the transformation to be equivalent, we apply it on a new rule of an existing relation.

```
assume(v ∈ G_P.get_all_nodes())

k = G_P.get_max_rule_number(v) + 1
args = G_P.generate_vars(v, k)
evals = {number : k, sign : +, vars : args}
G_{P_EQU} = G_P.R_AddRelEdge(v, v, evals)

assert(out(G_P) ≡ out(G_{P_EQU}))
```

EQU-FactInline. This transformation removes all incoming edges to a relational node v. Removing these edges effectively removes all rules for relation $R = \theta^{-1}(v)$. For the output of P_{EQU} to remain equivalent to the output of P, the removed rules are replaced with the corresponding facts that these rules would compute—we retrieve these facts by executing the rules for R. In particular, the transformation creates a fact node for each retrieved fact and associates it with v using the AddFact rewrite rule.

```
assume(v ∈ G_P.get_all_nodes())

E = G_P.get_incoming_edges(v)
G_{P_EQU} = G_P
foreach e ∈ E : G_{P_EQU} = G_{P_EQU}.R_DelRelEdge(e)
F = G_P.get_facts(v)
foreach f ∈ F : G_{P_EQU} = G_{P_EQU}.R_AddFact(f, v)
G_{P_EQU} = G_{P_EQU}.annotate()

assert(out(G_P) ≡ out(G_{P_EQU}))
```

At the end of this radical transformation, we re-annotate the resulting graph since all ancestors of v might now have different *stratum* and *ancestry* values.

CON-AddRelEdge. This is a contracting transformation that adds a positive relational edge e from u to v, where v is in positive ancestry of output node O and u either has the same stratification number as v or is not an ancestor of v at all (see Fig. 7 for an example). This is to ensure that e does not introduce a cycle with negation between u and v, thus rendering the transformed program unstratifiable. Adding an incoming edge to v corresponds to adding an atom in any rule for relation $R = \theta^{-1}(v)$, which contracts the result of R—adding an atom is essentially a conjunction in Datalog.

Since data flows monotonically from R to the output relation, this transformation will also contract the result of the program.

$\texttt{assume}(v \in \overline{G_P}.\texttt{get\_nodes\_with\_ancestry}(+) \bigwedge$
$\quad u \in \overline{G_P}.\texttt{get\_nodes\_with\_stratum}(v) \bigcup$
$\quad\quad (\overline{G_P}.\texttt{get\_all\_nodes}() \setminus \overline{G_P}.\texttt{get\_ancestors}(v)))$

$k = \texttt{generate\_number}(1, \overline{G_P}.\texttt{get\_max\_rule\_number}(v))$
$args = \overline{G_P}.\texttt{generate\_vars}(v, k)$
$evals = \{number : k,\ sign : +,\ vars : args\}$
$s = \overline{G_P}.\texttt{get\_stratum}(v)$
$a = \overline{G_P}.\texttt{get\_ancestry}(u)$
$nvals = \{stratum : s,\ ancestry : a\}$
$\overline{G_{P_{CON}}} = \overline{G_P}.\mathcal{R}_{\text{AddRelEdge}}(u, v, evals)$
$\overline{G_{P_{CON}}} = \overline{G_{P_{CON}}}.\mathcal{R}_{\text{ModRelNode}}(u, nvals)$

$\texttt{assert}(\texttt{out}(\overline{G_P}) \supseteq \texttt{out}(\overline{G_{P_{CON}}}))$

EXP-AddRelEdge. This is an expanding transformation that adds a positive relational edge e from u to v (see Fig. 8 for an example). Similar to the previous transformation, v is in positive ancestry of output node O, and u either has the same stratification number as v or is not an ancestor of v at all. Contrary to the previous transformation, adding e creates a new rule for relation $R = \theta^{-1}(v)$, which expands the result of R—adding a rule is essentially a disjunction in Datalog. Consequently, the program result is also expanded.

$\texttt{assume}(v \in \overline{G_P}.\texttt{get\_nodes\_with\_ancestry}(+) \bigwedge$
$\quad u \in \overline{G_P}.\texttt{get\_nodes\_with\_stratum}(v) \bigcup$
$\quad\quad (\overline{G_P}.\texttt{get\_all\_nodes}() \setminus \overline{G_P}.\texttt{get\_ancestors}(v)))$

$k = \overline{G_P}.\texttt{get\_max\_rule\_number}(v) + 1$
$args = \overline{G_P}.\texttt{generate\_vars}(v, k)$
$evals = \{number : k,\ sign : +,\ vars : args\}$
$s = \overline{G_P}.\texttt{get\_stratum}(v)$
$a = \overline{G_P}.\texttt{get\_ancestry}(u)$
$nvals = \{stratum : s,\ ancestry : a\}$
$\overline{G_{P_{EXP}}} = \overline{G_P}.\mathcal{R}_{\text{AddRelEdge}}(u, v, evals)$
$\overline{G_{P_{EXP}}} = \overline{G_{P_{EXP}}}.\mathcal{R}_{\text{ModRelNode}}(u, nvals)$

$\texttt{assert}(\texttt{out}(\overline{G_P}) \subseteq \texttt{out}(\overline{G_{P_{EXT}}}))$

---

6 IMPLEMENTATION

We implemented DLSmith in a total of 6,300 lines of Python code. It currently supports six Datalog dialects, namely, Ascent [45], DDlog [44], Flix [33], Formulog [4], Scallop [23], and Soufflé [24]. In the rest of this section, we discuss how to implement new metamorphic transformations as well as the existing queryFuzz transformations [34] in DLSmith.

Implementing new transformations. The transformations described in the previous section require (on average) 40 lines of Python code to implement. Implementing a transformation for an already supported Datalog engine involves the following steps: (1) expressing a precondition, (2) retrieving the graph elements satisfying the precondition, (3) generating attributes for the graph rewrite rules(s), (4) calling the graph rewrite rule(s), and (5) expressing a postcondition. Implementing a transformation for a new engine additionally requires extracting relations from seed programs and generating programs from annotated precedence graphs.

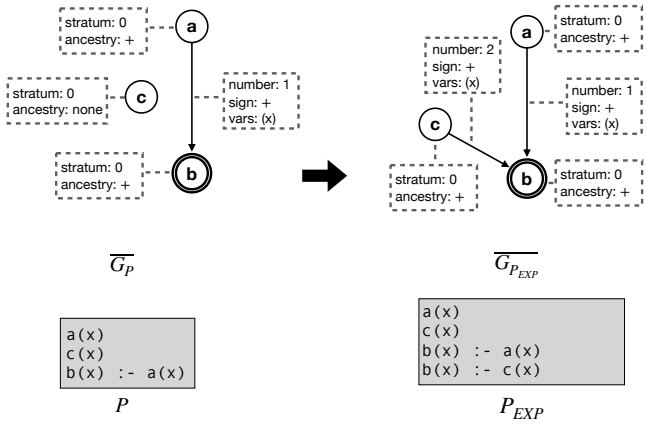


Figure 8: An example EXP-AddRelEdge transformation.

Implementing queryFuzz transformations. queryFuzz implements metamorphic transformations based on formal properties of conjunctive queries, namely, query containment and equivalence. As described earlier however, these transformations are limited since the approach does not have a global view of the program being transformed. On the other hand, DLSmith subsumes queryFuzz—not only can all queryFuzz transformations be expressed using the specifications in Sect. 5, but its transformations can now also be applied in *any* stratum of the Datalog program. In particular, this is how queryFuzz transformations can be expressed in DLSmith:

- Add transformations add an atom $R(v_1, \ldots, v_n)$ to a rule of relation Q. These can be expressed in DLSmith by adding a relational edge e from a relational node u to a relational node v, where $u = \theta(R)$ and $v = \theta(Q)$, using rewrite rule $\mathcal{R}_{\text{AddRelEdge}}$.
- Mod transformations modify a rule of relation Q by renaming a variable appearing in its atoms. These can be expressed in DLSmith by modifying property *vars* of the incoming edges to a node v, where $v = \theta(R)$, using rewrite rule $\mathcal{R}_{\text{ModRelEdge}}$.
- Rem transformations remove an atom $R(v_1, \ldots, v_n)$ from a rule of relation Q. These can be expressed in DLSmith by removing a relational edge e from a relational node u to a relational node v, where $u = \theta(R)$ and $v = \theta(Q)$, using rewrite rule $\mathcal{R}_{\text{DelRelEdge}}$.
- Neg transformations replace an atom $R(v_1, \ldots, v_n)$ in a rule of relation Q with the negated atom of a new relation, say *neg*. For instance, the following rule

```
p(X, Y) :- a(X, Y), b(Y, Z), c(Z).
```

is transformed into

```
neg(Z) :- a(X, Y), b(Y, Z), not c(Z).
p(X, Y) :- a(X, Y), b(Y, Z), not neg(Z).
```

Relation *neg* is defined to have the same body as the rule for Q but with a negated R, thereby introducing double negation. These transformations are always equivalent and may be expressed in DLSmith by generating a new relational node for *neg*, adding edges for the atoms of *neg*, and adjusting the edges for the atoms of Q.

7 EXPERIMENTAL EVALUATION

In this section, we address the following research questions:

RQ1: How effective is DLSmith in detecting previously unknown query bugs in diverse Datalog engines?
RQ2: What are characteristics of the detected bugs?
RQ3: How effective is DLSmith in terms of code coverage?
RQ4: How efficient is DLSmith?

7.1 Setup

We tested six mature Datalog engines, namely, Ascent, DDlog, Flix, Formulog, Scallop, and Soufflé. All engines are publicly available on GitHub. We completed the implementation of the first version of DLSmith in January 2022 and started our testing campaign with Soufflé. Until September 2022, we added support for the remaining five engines as well as for more transformations. On average, we spent about 1.5 months testing each engine.

As seeds, we used semantically valid test cases from the engine repositories. We employed 24 seeds for Ascent, 246 for DDlog, 169 for Formulog, 39 for Scallop, and 240 for Soufflé. Note that we did not use any seeds for Flix—Flix is a comprehensive functional programming language, and DLSmith does not currently support parsing Flix programs. Unless stated otherwise, for each seed program, DLSmith generates 500 transformed programs. To obtain each transformed program, DLSmith applies a random number of transformations, between 1 and 100.

We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30GHz machine with 256GB of memory, running Debian GNU/Linux 10 (buster).

7.2 Results

We now discuss our findings for each research question.

RQ1: Query bugs in diverse engines. We tested six active Datalog implementations, each supporting a different dialect. We give a brief overview of these engines next.

Ascent can integrate with arbitrary application logic written in the Rust programming language. In particular, it allows Datalog rules to call into Rust code and vice versa.
DDlog is used for incremental computation. Specifically, developers declaratively specify a desired input-output mapping, and DDlog uses it to synthesize an efficient incremental implementation.
Flix is a functional, imperative, and logic programming language, which looks like Scala and provides support for algebraic data types, pattern matching, higher order functions, etc. In Flix, Datalog programs are first class values, and Datalog constraints have more expressive power.
Formulog is a domain-specific dialect with support for constructing and reasoning about SMT formulas.

Table 2: Query bugs detected by DLSmith.

Bug ID	Datalog Engine	Metamorphic Transformation	Bug Status
1	Soufflé	EQU-AddRelEdges	Fixed
2	Soufflé	EQU-AddRelEdges	Fixed
3	Soufflé	EQU-AddRelEdges	Fixed
4	Soufflé	AddEqu	Fixed
5	Soufflé	AddEqu	Fixed
6	Soufflé	EQU-AddRelEdges	Confirmed
7	Formulog	EQU-AddSelfEdge	Fixed
8	Ascent	AddEqu	Fixed
9	Scallop	CON-AddRelEdge	Fixed
10	Scallop	CON-AddRelEdge	Fixed
11	Scallop	EXP-AddRelEdge	Fixed
12	Scallop	CON-AddRelEdge	Fixed
13	Scallop	EQU-AddRelEdges	Confirmed
14	Soufflé	EQU-AddRelEdges	Confirmed
15	Soufflé	EQU-FactInline	Confirmed
16	Soufflé	EQU-AddRelNode, EQU-AddRelEdges	Confirmed

Scallop is a Datalog-based neuro-symbolic programming language, supporting discrete, probabilistic, and differential reasoning modes. Rules may be integrated with machine-learning models, facts may have associated probabilities, results may be computed with a success probability, etc.
Soufflé is a fast and scalable dialect, whose syntax was inspired by bddbddb [53] and μZ in Z3 [21]. Its primary goal is speed, thereby tailoring program execution to multi-core servers with large memory.

Tab. 2 shows the list of unique and previously unknown query bugs detected by DLSmith. The first column of the table provides an identifier for each bug and links to the (anonymized) bug report on GitHub. The second column shows the engine where the bug was found, the third the metamorphic transformation that was applied, and the last column the status of the bug. In total, DLSmith detected 16 query bugs in four engines, all of which are confirmed by the developers and eleven are fixed.

Note that AddEqu (bugs 4, 5, 8) is a queryFuzz transformation implemented in DLSmith. Out of all detected bugs, only bugs 4 and 8 could have been detected by queryFuzz. Bug 5 is detected with a queryFuzz transformation, which however is not applied at the highest stratum—this is only possible in DLSmith. Also note that our tool applies sequences of transformations, and bug 16 required a sequence of two transformations to be detected.

RQ2: Characteristics of detected bugs. To better understand the characteristics of the detected bugs, we now discuss them in detail.

Soufflé. Bugs 1, 2, 5, and 15 were found in the implementation of the `eqrel` (equivalence relation) data structure. According to the developers, they recently applied performance-related changes to this code, which is perhaps when these bugs were introduced. Bug 3 was due to the incorrect implementation of utility function `range` in the presence of unsigned bounds. Bug 4 was caused by a mistake in the implementation of subsumption for the `btree_delete` data

structure. The developers called it a "nasty bug to find and fix". Bug 6 was detected in the code generation mechanism for the interpreter. Currently, the interpreter checks floating point number equivalence via bitwise comparison rather than floating point comparison. The developers confirmed the issue but need time to resolve it. Bug 14 was again caused by floating point equivalence checking in the `brie` data structure. This bug, however, only manifested in compiler mode. Bug 16 was also found in `brie`; it manifested when using "auto-scheduling" for automatic performance tuning.

Formulog. Bug 7 was a non-deterministic query bug in the procedure for computing strata, which incorrectly depended on non-deterministic hash values. As a result, relations ended up being computed in the wrong order.

Ascent. Bug 8 occurred because of an atom having a repeated variable in a rule body. This case was not taken into account when reordering atoms at runtime to increase performance.

Scallop. Bugs 9 and 11 were due to incorrect optimization conditions in the query plan optimizer. Bug 10 was related to incorrect variable deduplication in the query plan generator. Bug 12 revealed an issue with incorrect optimization of negative atoms, and bug 13 was caused by incorrectly detecting a negative cycle between two relations.

RQ3: Code coverage. In this research question, we evaluate the code coverage achieved by DLSmith. We first compare it with the coverage achieved by hand-crafted tests in the engine repositories, which we use as seeds for DLSmith (see Sect. 7.1). We also compare with queryFuzz when using the same seeds and with DLSmith when using empty seeds. When using empty seeds, phase 1 of Fig. 2 generates a random precedence graph, which is then passed on to phase 2 to generate a Datalog program. Hence, in case of an empty seed, DLSmith generates a Datalog program from scratch. The results are shown in Tab. 3 for Soufflé (written in C++) and Scallop (written in Rust), where we detected the most bugs. Note that when running queryFuzz, we use the same settings as for DLSmith, and that all results are averages computed over three runs.

As shown in the table, DLSmith is the most effective, while DLSmith-Empty (with empty seeds) is the least effective. When comparing DLSmith to running the seeds alone for Scallop, we observe a 4.8% and 7.5% increase in line and function coverage, respectively. For Soufflé, we observe a 4.1% and 2.8% increase in line and function coverage, respectively. When comparing DLSmith to queryFuzz for Scallop, we observe a 3.9% and 6.7% increase in line and function coverage, respectively. For Soufflé, we observe a 2.7% and 2.3% increase in line and function coverage, respectively.

RQ4: Performance. The performance of DLSmith depends on the Datalog engine under test. At the end of the second phase in Fig. 2, DLSmith on average generates a program per 0.006 seconds (or 163 programs per second). However, as expected, it is slowed down by the engine running each of these programs, and Tab. 4 shows by how much. The second column computes the average time of generating *and* running a single program (i.e., executing the first two phases of Fig. 2). The graph transformer in the third phase of Fig. 2 on average generates a transformed annotated precedence graph per 0.00035 seconds (or 2857 transformed annotated precedence graphs per second). Note that these results are averages computed over three runs of DLSmith, each seeded with 500 empty programs.

Table 3: Code coverage achieved by seeds alone, queryFuzz, DLSmith with empty seeds, and DLSmith. L represents line coverage, and F function coverage.

Datalog Engine	Seeds		queryFuzz	
	L	F	L	F
Scallop	18,056	2,709	18,201	2,729
Soufflé	63,452	40,350	64,298	40,544
	DLSmith-Empty		DLSmith	
	L	F	L	F
Scallop	11,388	1,826	18,915	2,912
Soufflé	50,180	30,205	66,027	41,484

Table 4: Average running time (in seconds) of DLSmith when executing its first two phases.

Datalog Engine	Running Time
Ascent	17.221
DDlog	612.121
Flix	240.031
Formulog	1.512
Scallop	0.146
Soufflé	0.734

7.3 Threats to Validity

Our experimental results, and especially the detected query bugs, depend on the Datalog engines we tested as well as the seed programs that DLSmith takes as input. Regarding the former, we selected six diverse and active Datalog implementations. Regarding the latter, we used all (syntactically and semantically) valid test cases from the engine repositories as seeds for DLSmith. We also perform an experiment using only empty seeds to show their effect on our code coverage results.

8 RELATED WORK

In this paper, we presented the most comprehensive approach to detecting query bugs in Datalog engines. Our approach uses metamorphic testing to address the oracle problem [3] by first generating a Datalog program from its corresponding annotated precedence graph and transforming the program by transforming its annotated precedence graph using graph rewrite rules.

Graphs are a powerful and general notation that is used to express and model complex systems in a variety of areas in computer science, including software engineering [22], software security [8], program slicing [14], computer networks [37], and bioinformatics [40], to name a few. Graph rewriting [42] is used to formalize how a complex structure represented by a graph evolves over time. Together with graph transformation [1, 2, 41], it has been extensively studied in the graph theory community. Next, we discuss the three most closely related areas of work.

Metamorphic testing. Metamorphic testing [15, 47] is a technique to effectively address the oracle problem in software testing. It works by constructing new test cases via mutating existing ones using metamorphic relations. These relations are then used to infer the output of the newly generated test cases. In previous work, we

presented the first metamorphic testing approach to detect bugs in Datalog engines [34]. Metamorphic testing has also been successfully used to test a variety of software applications [12, 27, 47, 57], including other query-based systems [43, 46, 63].

Testing compilers. Ensuring the correctness of compilers is a critical area of research, and significant effort has been devoted to testing compilers [13, 29, 48, 58]. Csmith [58] is a popular random program generator for C compilers that can generate large and complex programs. Le *et al.* [27] proposed a metamorphic testing approach to generate equivalent programs. The technique inspired a number of related approaches [17, 28, 29, 49, 62] that were used to detect hundreds of bugs in popular and widely used compilers. YarpGen [32] is a technique for generating expressive programs without undefined behavior to test C and C++ compilers. Recently, such techniques have also been extended to compilers for specialized domains, such as deep learning [31, 56] and quantum computers [39].

Testing program analyzers. Over the past years, program analyzers are becoming increasingly practical and are being widely adopted to ensure reliability of critical software systems. Work on automatically detecting bugs in program analyzers [11] and program analysis components is, therefore, receiving increased attention. For example, there have recently emerged techniques for testing model checkers [26, 61], SMT solvers [5, 9, 35, 52, 54, 55, 59, 60], symbolic execution engines [25], implementations of abstract domains [10], and dataflow analyses [50].

9 CONCLUSION

We have presented DLSmith, a novel approach for detecting query bugs in Datalog engines using dependency-aware metamorphic test oracles. Unlike existing, more limited, metamorphic oracles, our test oracles use rich precedence information capturing dependencies among relations in the program. DLSmith detected 16 previously unknown query bugs in four Datalog engines, and our evaluation showed that only two of these bugs could have been detected using existing techniques.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful and constructive feedback. This work was supported by Maria Christakis' ERC Starting grant 101076510.

REFERENCES

[1] Paolo Baldan, Andrea Corradini, and Barbara König. 2001. A Static Analysis Technique for Graph Transformation Systems. In *CONCUR (LNCS, Vol. 2154)*. Springer, 381–395.
[2] Paolo Baldan and Barbara König. 2002. Approximating the Behaviour of Graph Transformation Systems. In *ICGT (LNCS, Vol. 2505)*. Springer, 14–29.
[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *TSE* 41 (2015), 507–525. Issue 5.
[4] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis. In *OOPSLA*. ACM, 141:1–141:31.
[5] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *CAV (LNCS, Vol. 10982)*. Springer, 45–51.
[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *OOPSLA*. ACM, 243–262.
[7] Neville Brent, Lexiand Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *PLDI*. ACM, 454–469.
[8] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-Mutating Malware Using Control-Flow Graph Matching. In *DIMVA (LNCS, Vol. 4064)*. Springer, 129–143.
[9] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. ACM, 1459–1470.
[10] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.
[11] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the Program Analyser. In *ICSE*. ACM, 765–768.
[12] W. K. Chan, S. C. Cheung, and Karl R. P. H. Leung. 2005. Towards a Metamorphic Testing Methodology for Service-Oriented Software Applications. In *QSIC*. IEEE Computer Society, 470–476.
[13] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surv.* 53 (2020), 4:1–4:36. Issue 1.
[14] J.-L. Chen, F.-J. Wang, and Y.-L. Chen. 1997. An Object-Oriented Dependency Graph for Program Slicing. In *TOOLS*. IEEE Computer Society, 121–130.
[15] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST–CS98–01. HKUST.
[16] Distributed and Parallel Systems Group at the University of Innsbruck. 2012. Insieme Compiler. http://insieme-compiler.org.
[17] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *PACMPL* 1 (2017), 93:1–93:29. Issue OOPSLA.
[18] Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *Security*. USENIX, 1075–1092.
[19] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *PACMPL* 2 (2018), 116:1–116:27. Issue OOPSLA.
[20] Sergio Greco and Cristian Molinaro. 2015. *Datalog and Logic Databases.* Morgan & Claypool.
[21] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. 2011. μZ—An Efficient Engine for Fixed Points with Constraints. In *CAV (LNCS, Vol. 6806)*. Springer, 457–462.
[22] Susan Horwitz and Thomas W. Reps. 1992. The Use of Program Dependence Graphs in Software Engineering. In *ICSE*. ACM, 392–411.
[23] Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. 2021. Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning. In *NeurIPS*. 25134–25145.
[24] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *CAV (LNCS, Vol. 9780)*. Springer, 422–430.
[25] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *ASE*. IEEE Computer Society, 590–600.
[26] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.
[27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*. ACM, 216–226.
[28] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*. ACM, 386–399.
[29] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *PLDI*. ACM, 65–76.
[30] Vladimir Lifschitz. 1988. On the Declarative Semantics of Logic Programs with Negation. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 177–192.
[31] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2022. Finding Deep-Learning Compilation Bugs with NNSmith. *CoRR* abs/2207.13066 (2022).
[32] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *PACMPL* 4 (2020), 196:1–196:25. Issue OOPSLA.
[33] Magnus Madsen and Ondrej Lhoták. 2018. Safe and Sound Program Analysis with FLIX. In *ISSTA*. ACM, 38–48.
[34] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic Testing of Datalog Engines. In *ESEC/FSE*. ACM, 639–650.
[35] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE*. ACM, 701–712.
[36] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.
[37] John M. McQuillan. 1977. Graph Theory Applied to Optimal Connectivity in Computer Networks. *Comput. Commun. Rev.* 7 (1977), 13–41. Issue 2.
[38] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *PLDI*. ACM, 308–319.
[39] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum Computing Platforms: An Empirical Study. *PACMPL* 6 (2022), 1–27. Issue OOPSLA.

[40] Georgios A. Pavlopoulos, Maria Secrier, Charalampos N. Moschopoulos, Theodoros G. Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G. Bagos. 2011. Using Graph Theory to Analyze Biological Networks. *BioData Min.* 4 (2011). Issue 10.
[41] Karl-Heinz Pennemann. 2008. Development of Correct Graph Transformation Systems. In *ICGT (LNCS, Vol. 5214)*. Springer, 508–510.
[42] Detlef Plump. 1995. On Termination of Graph Rewriting. In *WG (LNCS, Vol. 1017)*. Springer, 88–100.
[43] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *PACMPL* 4 (2020), 211:1–211:30. Issue OOPSLA.
[44] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog (CEUR, Vol. 2368)*. CEUR-WS.org, 56–67.
[45] Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. 2022. Seamless Deductive Inference via Macros. In *CC*. ACM, 77–88.
[46] Sergio Segura, Juan C. Alonso, Alberto Martin-Lopez, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. 2022. Automated Generation of Metamorphic Relations for Query-Based Systems. In *MET@ICSE*. ACM, 48–55.
[47] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *TSE* 42 (2016), 805–824. Issue 9.
[48] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *ICSE*. ACM, 203–213.
[49] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*. ACM, 849–863.
[50] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.
[51] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS*. ACM, 67–82.
[52] Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. 2020. TestMC: Testing Model Counters Using Differential and Metamorphic Testing. In *ASE*. IEEE Computer Society, 709–721.
[53] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *APLAS (LNCS, Vol. 3780)*. Springer, 97–118.
[54] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *PACMPL* 4 (2020), 193:1–193:25. Issue OOPSLA.
[55] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI*. ACM, 718–730.
[56] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proc. ACM Meas. Anal. Comput. Syst.* 6 (2022), 15:1–15:28. Issue 1.
[57] Xiaoyuan Xie, Joshua Wing Kei Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. 2009. Application of Metamorphic Testing to Supervised Classifiers. In *QSIC*. IEEE Computer Society, 135–144.
[58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. ACM, 283–294.
[59] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *ISSTA*. ACM, 322–335.
[60] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal Approximation Enumeration for SMT Solver Testing. In *ESEC/FSE*. ACM, 1141–1153.
[61] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.
[62] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*. ACM, 347–361.
[63] Zhiquan Zhou, Shaowen Xiang, and Tsong Yueh Chen. 2016. Metamorphic Testing for Software Quality Assessment: A Study of Search Engines. *TSE* 42 (2016), 264–284. Issue 3.

Received 2023-02-16; accepted 2023-05-03