Synthesizing Code Quality Rules from Examples

PRANAV GARG, Amazon Web Services, USA
SRINIVASAN H. SENGAMEDU, Amazon, USA

Static Analysis tools have rules for several code quality issues and these rules are created by experts manually. In this paper, we address the problem of automatic synthesis of code quality rules from examples. We formulate the rule synthesis problem as synthesizing first order logic formulas over graph representations of code. We present a new synthesis algorithm RhoSynth that is based on Integer Linear Programming-based graph alignment for identifying code elements of interest to the rule. We bootstrap RhoSynth by leveraging code changes made by developers as the source of positive and negative examples. We also address rule refinement in which the rules are incrementally improved with additional user-provided examples. We validate RhoSynth by synthesizing more than 30 Java code quality rules. These rules have been deployed as part of Amazon CodeGuru Reviewer and their precision exceeds 75% based on developer feedback collected during live code-reviews within Amazon. Through comparisons with recent baselines, we show that current state-of-the-art program synthesis approaches are unable to synthesize most of these rules.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Programming by example**.

Additional Key Words and Phrases: Program Synthesis

ACM Reference Format:
Pranav Garg and Srinivasan H. Sengamedu. 2022. Synthesizing Code Quality Rules from Examples. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 187 (October 2022), 31 pages. https://doi.org/10.1145/3563350

1 INTRODUCTION

Software is an integral part of today's life and low code quality has large costs associated with it. For example, the Consortium for Information & Software Quality notes that "For the year 2020, the total Cost of Poor Software Quality (CPSQ) in the US is $2.08 trillion."[1] Hence automated tools to detect code quality issues are an important part of software development. Developers use a variety of tools to improve their software quality, ranging from linters (e.g., SonarLint) to general-purpose code analysis tools (e.g., FindBugs, ErrorProne and Facebook Infer) to specialized tools (e.g., FindSecBugs for code security). There are many facets to code quality and often developers want to develop custom analyzers which detect code quality issues that are of specific interest to them.

A common approach for developing custom analyzers or checkers is expressing them in a domain specific language (DSL), such as Semmle's CodeQL[2] or Semgrep[3], or Amazon's GQL [Mukherjee et al. 2022]. Typically, code analysis tools such as SonarQube or Semgrep have thousands of rules that have been manually developed by the community of its users or the product owners. Development of these rules is a continuous process: new rules are added to guide usage of new APIs or frameworks, and existing rules evolve as the APIs evolve. Further, expressing code checks

[1] https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf
[2] https://codeql.github.com/
[3] https://semgrep.dev/

Authors' addresses: Pranav Garg, Amazon Web Services, 7 W 34th St., New York, NY, 10001, USA, prangarg@amazon.com; Srinivasan H. Sengamedu, Amazon, 2250 7th Ave, Seattle, WA, 98121, USA, sengamed@amazon.com.

2475-1421/2022/10-ART187
https://doi.org/10.1145/3563350

in a DSL requires specialized knowledge and extensive testing. On the whole, the development of rules is tedious and expensive, and the amount of required human intervention limits the scalability of this manual approach.

In this paper, we address the problem of developing code quality rules from labeled code examples. These examples can be provided by rule authors in the form of conforming and violating examples. In most cases, providing these examples is easier than writing the rule itself. In fact, often, these rules are developed following test-driven development (TDD) [Beck 2002], where the conforming and violating code examples are used to guide the development of these rules [Ryzhkov 2011]. Rules from various tools such as SonarQube and Semgrep, actually, come with such labeled test code examples. Additionally, we can leverage *code changes* as the source of labeled examples. In a corpus of code changes, common code quality issues will have multiple code changes that fix those issues. Similar code changes that are performed by multiple developers across projects can be used to obtain examples to synthesize code quality rules. For a code change (`code-before`, `code-after`) that fixes a code quality issue, `code-before` is a violating example and `code-after` is a conforming example.

We propose RhoSynth, an algorithm for automatically synthesizing high-precision rules from labeled code examples. RhoSynth reduces the rule synthesis problem to synthesizing first order logic formulas over Program Dependence Graphs (PDGs). Using a PDG based program representation not only allows our approach to be more robust to syntactic variations in code, but it also enables a succinct expression of semantic information such as data-flow relation and control-dependence. Compared to existing approaches that synthesize bug-fixes on Abstract Syntax Trees (ASTs) [Bader et al. 2019; Rolim et al. 2017, 2018], synthesizing rules over a graph representation comes with its own challenge. Unlike trees, where a node can be uniquely identified with the path from the root of the tree to itself, there is no such unique id for nodes in a graph. This ambiguity in identifying a node, consequently identifying a subgraph, makes the synthesis problem hard [Haussler 1989]. We use Integer Linear Programming (ILP) [Lerouge et al. 2017; Schrijver 1986] to solve this problem that lies at the core of rule synthesis.

We express rules as first order logic formulas comprising an existentially quantified *buggy pattern* and an existentially quantified *non-buggy pattern*. Specifically, rules have the following format: $\exists\vec{x}.bp(\vec{x}) \wedge \neg\big(\bigvee_i \exists\vec{y}.nbp_i(\vec{x},\vec{y})\big)$, where $\vec{x}$ and $\vec{y}$ denotes a set of nodes in the PDG. The buggy pattern captures the applicability of the rule in violating code examples and the non-buggy pattern captures the pattern that *must* be present in conforming code, if the code example satisfies the buggy pattern. This is a rich format that can express a wide range of code quality issues. We synthesize such rules by first synthesizing a buggy pattern $\exists\vec{x}.bp(\vec{x})$ over violating examples, and then synthesizing the non-buggy pattern $\exists\vec{y}.nbp_i(\vec{x},\vec{y})$ that accepts conforming examples satisfying $bp(\vec{x})$ for a given valuation $\vec{x}$. Through this decomposition, we simplify the rule synthesis problem to synthesizing existentially quantified buggy and non-buggy patterns, which are themselves synthesized using ILP. Further, the non-buggy pattern may contain disjunctions. We propose a top-down entropy-based algorithm to partition conforming examples into groups and synthesize a conjunctive non-buggy disjunctive pattern for each group.

Recently, there has been lots of work on automated program repair from code changes [Bader et al. 2019; Bavishi et al. 2019; Meng et al. 2013; Rolim et al. 2017, 2018]. While code changes themselves are a good source of examples for synthesizing code quality rules, there may be variations in correct code that are not captured in code changes. We propose *rule refinement* to incrementally improve the rule by providing additional examples corresponding to such code variations. It turns out, both ILP-based graph alignment and disjunctive non-buggy pattern synthesis are instrumental in leveraging "unpaired" code examples for refining the rules.

The main contributions of the paper are as follows.

(1) We formulate the problem of synthesizing rules from labeled code examples as synthesis of logical formulas over graph representations of code. We present a novel algorithm RhoSynth that is based on ILP based graph alignment, for identifying nodes in the graph that are relevant to the rule being synthesized.
(2) We propose *rule refinement* which improves the precision of rules based on a small number of labeled false positive examples.
(3) We validate our algorithm by synthesizing more than 30 Java code-quality rules from labeled code examples obtained from code changes in GitHub packages. These rules have been deployed as part of Amazon CodeGuru Reviewer. We validate these synthesized rules based on offline evaluation as well as live code review feedback collected over a period of several months at Amazon. The precision of synthesized rules exceeds 75% in production. Rule refinement improves the precision of rules by as much as 68% in some cases.
(4) We show that recent program synthesis baselines can synthesize only 22% to 61% of the rules. In addition, we show that, compared to ILP-based graph alignment, commonly used tree-differencing approaches do not perform well when aligning unpaired code examples for rule refinement and this results in rules not being synthesized in a majority of cases.

The paper is organized as follows. Section 2 formally defines the rule synthesis problem and provides an outline of the approach. Section 3 describes the representation for programs and rules. Section 4 describes the rule synthesis algorithm. Section 5 describes the implementation details. Section 6 describes the experimental results. Section 7 presents related work and Section 8 concludes the paper. Supplementary Appendix contains details such as proofs and examples.

2 RHOSYNTH OVERVIEW

This section precisely defines the problem and provides an overview of the approach with a running example.

2.1 Problem Statement

We are given a set of violating or buggy code examples $\mathcal{V} = \{V_1, \cdots, V_m\}$ and a set of conforming or non-buggy code examples $\mathcal{C} = \{C_1, \cdots, C_n\}$, for a *single code quality issue*. The problem is to synthesize a rule R from a subset of examples such that $R(V_i)$ = `True` and $R(C_j)$ = `False`, for all i and j in the held-out test set. We consider code examples at the granularity of a method. This offers sufficient code context to precisely capture a wide range of code quality issues [Sobreira et al. 2018; Tufano et al. 2018], while at the same time being simple enough to facilitate efficient rule synthesis.

The corpus of code changes made by developers is a natural source of positive and negative examples. In such a corpus, common code quality issues will have multiple code changes that fix those issues. It is possible to obtain examples for *single code quality issues* in an automated manner by clustering code changes [Bader et al. 2019; Kreutzer et al. 2016; Paletov et al. 2018]. Code changes in the input consist of pairs (B_i, A_i) where B_i is the code-before and A_i is the code-after. To synthesize a rule R, we consider `code-befores` as violating examples and `code-afters` as conforming examples.

A user can also provide additional *conforming examples* corresponding to variations in correct code that may not be captured by the code changes. We obtain such examples by identifying false positives in detections generated by the synthesized rule. This again is a natural way of obtaining examples. Another source of examples is false negatives. We do not consider this scenario since such examples are harder to obtain.

2.2 RhoSynth Steps

Consider the two code changes shown in Figure 1(a)-(b). The code before the change does not handle the case when the cursor accessing the result set of a database query is empty. Without this check, the app might crash when subsequent operations are called on the cursor (e.g., `getString` call on line 8 in Figure 1(a). In these code changes, the developer adds this check by handling the case when `Cursor.moveToFirst()` returns `False`.

(a)

```
1  ...
2  Cursor cursor = cr.query(...);
3- cursor.moveToFirst();
4+ if (!cursor.moveToFirst()) {
5+   cursor.close();
6+   return null;
7+ }
8  final String id =
       cursor.getString(0);
```

(b)

```
1  ...
2- mProviderCursor.moveToFirst();
3+ if (!mProviderCursor.moveToFirst()) {
4+   return;
5+ }
6  do {
7    if(mProviderCursor.getLong(...) ==
       id) {
8  ...
```

Fig. 1. Examples of input code changes.

As mentioned earlier, RhoSynth first synthesizes the buggy pattern from buggy code examples and then synthesizes the non-buggy pattern from non-buggy code examples.

Buggy Pattern Synthesis: The buggy pattern synthesis uses only buggy code examples, which correspond to the set of `code-befores` in the input code changes. We perform a graph alignment on their PDG representations to know the correspondence between nodes in different examples. The graph alignment is framed as an ILP optimization problem. In our example, graph alignment determines that the data variables `cursor` in Figure 1(a) and `mProviderCursor` in Figure 1(b) correspond to each other. Similarly, the calls `moveToFirst` correspond. On the other hand, call `getString` in the first example does not have any corresponding node in the second example. We use this node correspondence map to construct a Unified Annotated PDG (UAPDG) representation that encapsulates information from all buggy examples. Figure 2 partially illustrates this UAPDG $\mathcal{A}$. *The solid lines in the figure indicate that the corresponding nodes and edges are present in all buggy examples.* We project $\mathcal{A}$ to these solid nodes and edges and obtain $\mathcal{A}_c$, which is shown in Figure 4(a). $\mathcal{A}_c$ corresponds to the buggy pattern $\exists x_0, x_1.bp(x_0, x_1)$ where $bp(x_0, x_1)$ is described in Figure 4(d). Besides other checks, the buggy pattern asserts that the output of `moveToFirst` call is ignored, i.e., it is not defined or not used.

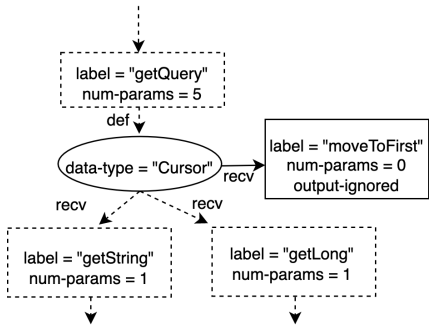


Fig. 2. Unified Annotated PDG (UAPDG) representation that captures *all* code-before's in the input. Dashed lines indicate that the corresponding nodes and edges are present in only a subset of the examples.

Non-buggy Pattern Synthesis: We now find all non-buggy code examples that satisfy the buggy pattern. We use a satisfiability solver for this. If there are such examples, we synthesize a non-buggy pattern (*nbp*) from them and strengthen the buggy pattern with $\neg nbp$ so that the overall rule rejects the non-buggy examples. In our running example, there are no `code-after` examples in the

```
1  ...
2  Cursor c = sql.query(...);
3  c.moveToFirst();
4  while (!c.isAfterLast()) {
5    Record record = cursorToCognitoRecord(c);
6    recordList.add(record);
7    c.moveToNext();
8  }
9  ...
```

(a)

```
1  ...
2  Cursor c = cr.query(...);
3  if (c.getCount() == 1) {
4    c.moveToFirst();
5    final String key = c.getString(...);
6    ...
7  } else {
8    Log.debug("...");
9    ...
10 }
```

(b)

Fig. 3. Non-buggy code examples used for refining the originally synthesized rule

input that satisfy the buggy pattern, since the value returned by `moveToFirst` is used in all of those examples. Consequently, we synthesize a vacuous non-buggy pattern = `False` and the overall rule is the same as the buggy pattern we synthesized above.

Rule Refinement: In several cases, the initial set of examples does not capture all code variations. In case of our current rule, the following checks also check the emptiness of the result set:

(1) `Cursor.getCount() == 0`.
(2) `Cursor.isAfterLast()` returns `True`.

These variations are not part of the initial examples. When we run the synthesized rule on code corpus, we encounter examples such as the ones shown in Figure 3(a)-(b) that check the cursor using these code variations. Note these examples are not accompanied by buggy code. We propose *rule refinement* that uses these additional non-buggy examples to improve the rule. Specifically, we re-synthesize the non-buggy pattern by constructing a UAPDG $\mathcal{A}_{pc}$, in a way similar to the UAPDG construction in buggy pattern synthesis. However, it turns out that $\mathcal{A}_{pc}$ is too general and accepts even the buggy examples. We use this as a forcing function to partition the non-buggy examples and synthesize a conjunctive non-buggy pattern for each partition.

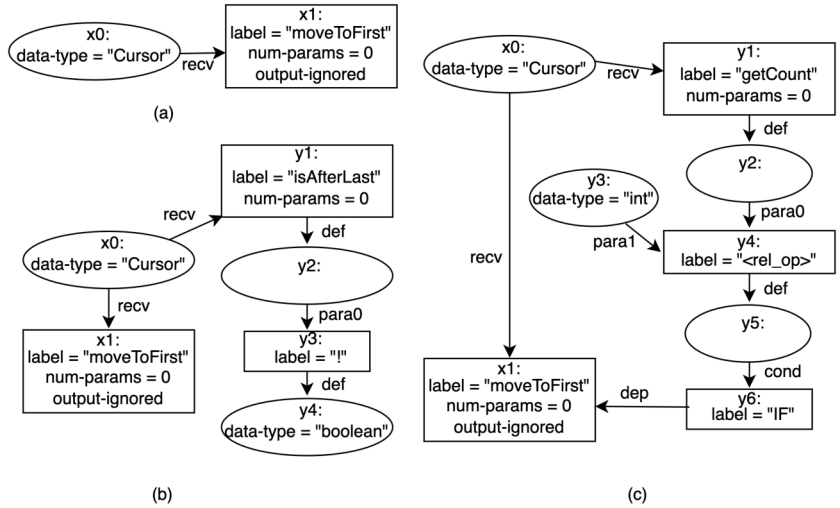
Figure 4(a)-(c) illustrate the UAPDGs for the synthesized buggy pattern and the two disjuncts in the non-buggy pattern. Figure 4(d) provides the exact rule, expressed as a logical formula. Informally, the synthesized rule R satisfies all code examples that call `Cursor.moveToFirst` such that they do not check the value returned by this call, nor call `Cursor.isAfterLast` or `Cursor. getCount`. When we run this rule again on the code examples, it correctly accepts all the buggy examples and rejects all non-buggy examples, including the additional examples that were used for rule refinement.

3 PROGRAM AND RULE REPRESENTATION

In this section, we describe the PDG based representation of code (Section 3.1), the rule syntax (Section 3.2) and introduce Unified Annotated PDGs (UAPDGs) for representing rules (Section 3.3).

3.1 Code Representation

We represent code examples at a method granularity using PDGs [Ferrante et al. 1987]. PDG is a labeled graph that captures all data and control dependencies in a program. Nodes in the PDG are classified as *data nodes* and *action nodes*. Data nodes are, optionally, labeled with the data types and values for literals, and action nodes are labeled with the operations they correspond to, for e.g., method name for method call nodes, etc. Edges in the PDG correspond to data-flow and control dependencies and are labeled as *recv* (connects receiver object to a method call node),



$$R = \exists x_0, x_1.\Big[bp(x_0, x_1) \ \wedge \neg\big(\exists y_1, \cdots, y_4.nbp_1(x_0, x_1, y_1, \cdots, y_4) \vee \exists y_1, \cdots, y_6.nbp_2(x_0, x_1, y_1, \cdots, y_6)\big)\Big], \ \textit{where}$$

$$bp(x_0, x_1) := \textit{data-type}(x_0) = \textit{"Cursor"} \wedge \textit{label}(x_1) = \textit{"moveToFirst"} \wedge \textit{num-para}(x_1) = 0 \wedge \textit{output-ignored}(x_1) \wedge x_0 \xrightarrow{recv} x_1$$

$$nbp_1(x_0, x_1, y_1, \cdots, y_4) := \textit{label}(y_1) = \textit{"isAfterLast"} \wedge \textit{num-para}(y_1) = 0 \wedge \textit{data-type}(y_2) = \textit{".*"} \wedge \textit{label}(y_3) = \textit{"!"} \wedge \textit{data-type}(y_4) = \textit{"boolean"} \wedge x_0 \xrightarrow{recv} y_1 \wedge y_1 \xrightarrow{def} y_2 \wedge y_2 \xrightarrow{para0} y_3 \wedge y_3 \xrightarrow{def} y_4$$

$$nbp_2(x_0, x_1, y_1, \cdots, y_6) := \textit{label}(y_1) = \textit{"getCount"} \wedge \textit{num-para}(y_1) = 0 \wedge \textit{data-type}(y_2) = \textit{".*"} \wedge \textit{data-type}(y_3) = \textit{"int"} \wedge \textit{label}(y_4) = \textit{"<rel\_op>"} \wedge \textit{data-type}(y_5) = \textit{".*"} \wedge \textit{data-type}(y_6) = \textit{"IF"} \wedge x_0 \xrightarrow{recv} y_1 \wedge y_1 \xrightarrow{def} y_2 \wedge y_2 \xrightarrow{para0} y_4 \wedge y_3 \xrightarrow{para1} y_4 \wedge y_4 \xrightarrow{def} y_5 \wedge y_5 \xrightarrow{cond} y_6 \wedge y_6 \xrightarrow{dep} x_1$$

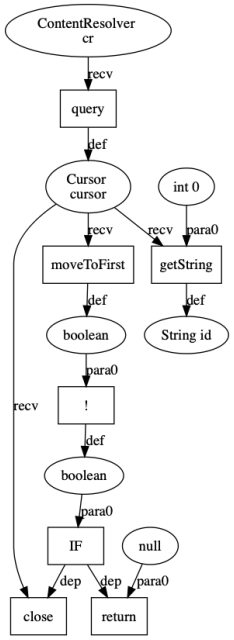
(d)

Fig. 4. Rule synthesized from code examples in Figure 1: (a) UAPDG for the buggy pattern (b) UAPDG for the first disjunct in the non-buggy pattern (c) UAPDG for the second disjunct in the non-buggy pattern (d) Overall rule, after refinement, expressed in logic.

$para_i$ (connects the i^{th} parameter to the operation), *def* (connects an action node to the data value it defines), *dep* (connects an action node to all nodes that are directly control dependent on it) and *throw* (connects a method call node to a `catch` node indicating exceptional control flow). See Figure 5a for the PDG representation of code-after in Figure 1(a).

3.2 Rule Syntax

In this work, we express rules as quantified first-order logic formulas over PDGs (refer to Figure 5b for a detailed syntax of rules). A rule is a formula of the form $\exists\vec{x}.bp(\vec{x}) \wedge \neg\big(\bigvee \exists\vec{y}.nbp(\vec{x}, \vec{y})\big)$, where $\vec{x}$ and $\vec{y}$ are a set of quantified variables that range over distinct nodes in a PDG. The buggy pattern $bp(\vec{x})$ evaluates to `True` on buggy code and the non-buggy pattern $nbp_i(\vec{x}, \vec{y})$ evaluates to `True` on correct code, with appropriate instantiations for $\vec{x}$, $\vec{y}$. Because of the negation before the non-buggy pattern, the entire formula evaluates to `True` on buggy code and `False` on correct code. Intuitively, the buggy pattern captures code elements of interest in buggy, and possibly correct, code, and



(a)

$$\text{Rule}: \quad R \stackrel{def}{=} \exists \vec{x}.bp(\vec{x}) \wedge \neg\big(\bigvee_i \exists \vec{y}.nbp_i(\vec{x},\vec{y})\big), \ \textit{where}$$
$$bp(\vec{x}) \in \varphi(\vec{x}), \quad \textit{and} \quad nbp_i(\vec{x},\vec{y}) \in \varphi(\vec{x},\vec{y})$$

Conjunctive Subrule: $\varphi(\vec{x}) ::= \bigwedge_i \varphi^i(\vec{x}) \mid \epsilon(\vec{x}) \mid \eta(\vec{x}) \mid \texttt{True} \mid \texttt{False}$

Edge predicate: $\epsilon(\vec{x}) ::= x_1 \xrightarrow{e} x_2$

Node predicate: $\eta(\vec{x}) ::= \ label(x) = l \mid data\text{-}type(x) = R$
$\mid data\text{-}value(x) = s \mid num\text{-}para(x) = i$
$\mid declaring\text{-}type(x) = R$
$\mid trans\text{-}control\text{-}dep(x) \supseteq 2^{Label}$
$\mid output\text{-}ignored(x) \mid \neg output\text{-}ignored(x)$

$\vec{x} = \{x_1, \cdots\}, \vec{y} = \{y_1, \cdots\}, x_i, y_i \in Var, l \in Label,$
$s \in String, i \in Int, R \text{ is regular expression},$
$e \in \{recv, para_i, def, dep, throw\}$

(b)

Fig. 5. (a) PDG for the snippet of code-after in Figure 1(a). Oval nodes indicate data nodes and rectangular nodes indicate action nodes. (b) Syntax of Rules.

the non-buggy pattern captures the same in correct code. The elements appear as existentially quantified variables. This is a rich format that can express a wide range of code quality issues.

Formulas $bp(\vec{x})$ and $nbp_i(\vec{x},\vec{y})$ are quantifier-free sub-rules comprising a conjunction of atomic edge predicates $\epsilon(\vec{x})$ that correspond to edges $x_1 \xrightarrow{e} x_2$ in the PDG, and atomic node predicates $\eta(\vec{x})$. $\eta(\vec{x})$ express various properties at PDG nodes including the node label, data-type, data values for literals, number of parameters for method calls (*num-para(x)*), declaring class type for static method calls (*declaring-type(x)*), the set of nodes on which x is transitively control dependent (*trans-control-dep(x)*) and, finally, whether a method call's output is/is not ignored (*output-ignored*(x) and its negation)[4].

Note, we exclude disjunctions in buggy patterns since a rule with a disjunctive buggy pattern can be expressed as multiple rules without a disjunctive buggy pattern, in the rule syntax. Further, a rule with a buggy pattern that is negated (e.g., $\neg x_1 \xrightarrow{e} x_2$) is expressed using a positive non-buggy pattern, and vice-versa. The rule syntax, which mostly captures first-order logic properties over PDGs, includes some higher-order properties, e.g., transitive control dependence. Our rule syntax is influenced by the Guru Query Language (GQL) [Mukherjee et al. 2022], which is an imperative, Java-based domain-specific language used at Amazon for creating code analysis rules.

Checking if a rule satisfies a code example represented as a PDG can be reduced to satisfiability modulo theories (SMT) [Barrett and Tinelli 2018]. Since PDGs are finite graphs, this satisfiability check is decidable. By mapping nodes in the PDG to bounded integers, this check can be

[4]Predicate *output-ignored*(x) is `True` for a method call when it does not return a value or the returned value has no users, and `False` otherwise.

reduced to satisfiability in the Presburger arithmetic. Moreover, state-of-the-art SMT solvers such as Z3 [De Moura and Bjørner 2008] can discharge these checks efficiently.

3.3 Rule Representation

Rules are formulas that accept or reject code examples represented as PDGs. From the rule syntax (Figure 5b), a rule is expressed using a buggy and non-buggy patterns. Both the buggy pattern and the non-buggy pattern can be expressed as a collection of quantified formulas $\mathcal{P}_Q = \exists \vec{y}.\varphi(\vec{x}, \vec{y})$, where φ is a conjunctive subrule defined over free variables $\vec{x}$ and bound variables $\vec{y}$. To represent rules, we need a representation for $\mathcal{P}_Q$. We arrive at a representation for $\mathcal{P}_Q$, from a PDG representation, in two steps. We first introduce Valuated PDGs (VPDGs; also, sometimes, referred as valuated examples) that are PDGs extended with variables. VPDGs capture a *single* PDG that is accepted by $\mathcal{P}_Q$. We then introduce Unified Annotated PDGs (UAPDGs) that accept *sets* of VPDGs and represent the quantified formula $\mathcal{P}_Q$.

Let *Var* be a set of variables. A Valuated PDG is a PDG with a subset of its nodes mapped to distinct variables in *Var*. Informally, a Valuated PDG is a PDG with a valuation for variables.

As mentioned above, a Unified Annotated PDG is defined over a set of free variables $Var_f \subseteq Var$ and bound variables $Var_b \subseteq Var$, and represents an existentially quantified conjunctive formula over PDGs of the form $\exists Var_b.\varphi(Var_f, Var_b)$. Note that φ is a conjunctive subrule that is defined over a set of node predicates ($\eta(\vec{x})$ in Figure 5b). These node predicates are expressed using join semi-lattices annotating different nodes in a UAPDG (see Section 5 for details). Formally,

Definition 1 (Unified Annotated PDG). Given a distinct set of free variables Var_f and bound variables Var_b, a *Unified Annotated PDG* $= (N, E, Lat : N \to (Lattice_1 \times \cdots \times Lattice_k), M_f : N \mapsto Var_f, M_b : N \mapsto Var_b)$, where $Lat(n)$ is the set of node predicates at n expressed using join semi-lattices, and M_f and M_b are maps from a disjoint subset of its nodes to distinct variables in Var_f and Var_b, respectively.

Note that bound variables Var_b can be permuted in the map M_b without affecting the semantics of a UAPDG. On the other hand, variables Var_f cannot be permuted or renamed in the map M_f, since these variables are free (they are bound to an outer existential quantifier in the rule R). We call nodes in M_f, mapped to variables in Var_f, *frozen* since the variables mapped to these nodes are fixed for a given UAPDG.

For constructing a UAPDG from a set of VPDGs, we need to first *identify corresponding nodes* in VPDGs and then *unify* the node predicates at the corresponding nodes. We identify corresponding nodes through graph alignment using ILP (see Section 4.1) and the unification of node predicates is performed by generalization over the lattice. We describe the use of UAPDG representation to synthesize buggy patterns and non-buggy patterns in Section 4.

Notation: We use italicized font to denote a single example or a PDG or a Valuated PDG (e.g., *C, E, V*) and script font to denote sets of examples or a Unified Annotated PDG (e.g., $\mathcal{A}$, $\mathcal{C}$, $\mathcal{E}$, $\mathcal{V}$). Also, we use superscripts, e.g., $\mathcal{A}^{\vec{x}}$, to denote the set of free variables over which a UAPDG or VPDG is defined ($\mathcal{A}^{\emptyset}$ denotes UAPDG with no free variables).

Example 2. Figure 4(b) illustrates a UAPDG defined over free variables $\{x_0, x_1\}$ and bound variables $\{y_1, y_2, y_3, y_4\}$. The label at each node in the figure visualizes the value of maps M_f, M_b and *Lat* at that node. In this UAPDG, let n_1 be the node with data type "Cursor" and n_2 be the node with label "moveToFirst". Then, at node n_1, $M_f(n_1) = x_0$, $Lat(n_1) = (\textit{data-type} = \textit{"Cursor"}, \textit{data-value} = \top, \cdots)$ and map M_b is undefined. Further, we say that nodes n_1 and n_2 are *frozen* in this UAPDG since they are mapped to free variables x_0 and x_1 respectively.

Translating a UAPDG to an existentially quantified formula over graphs: A UAPDG naturally translates into an existentially quantified conjunctive formula over Valuated PDGs. If Var_f is the set of free variables, and Var_b is the set of bound variables, the translation of the UAPDG is a formula $\exists Var_b.\varphi(Var_f, Var_b)$, where φ is the conjunction of all satisfying atomic node and edge predicates expressed over Var_f and Var_b.

Example 3. Refer to Figure 4(d) for the quantified rule that corresponds to the UAPDGs in Figure 4(a)-(c). Specifically, the UAPDG in Figure 4(b) is equivalent to the formula:
$\exists y_1, \cdots, y_4.nbp_1(x_0, x_1, y_1, \cdots, y_4)$.

Constructing a UAPDG from a Valuated PDG: We can construct a UAPDG from a Valuated PDG or a PDG, which can be seen as a Valuated PDG with an empty node-variable map, in the following manner:

- Inherit the set of nodes N and the set of edges E from the Valuated PDG.
- Construct the map Lat by iterating over all nodes n in the Valuated PDG and computing the lattice values for all node predicates at n.
- Initialize M_f with the node-variable map in the Valuated PDG.
- Construct M_b by mapping all nodes outside the domain of M_f to distinct variables in Var_b (these nodes are bound locally to an existential quantifier in the formula that captures the UAPDG).

Note that a UAPDG obtained by enhancing a PDG (or Valuated PDG), in the above way, corresponds to a formula that accepts the PDG (or Valuated PDG). Henceforth, in the text, we use UAPDG to refer to both the graph structure obtained by enhancing PDGs or Valuated PDGs and the existentially quantified formulas they corresponds to, interchangeably.

4 RULE SYNTHESIS ALGORITHM

The outline of the rule synthesis algorithm is as follows. We first synthesize the buggy pattern based on violating examples. We then synthesize the non-buggy pattern based on conforming examples that satisfy the buggy pattern. If the non-buggy pattern is not satisfied by any violating example, the synthesis is complete. If a violating example satisfies a non-buggy pattern, we follow a hierarchical partitioning approach: pick a conjunct from non-buggy patterns which is satisfied by the violating example, split the underlying partition of non-buggy examples based on an entropy-based algorithm and recursively synthesize non-buggy patterns for the new partitions. The final non-buggy pattern is the disjunction of non-buggy patterns synthesized over the leaf partitions.

The overall rule synthesis algorithm is described in Figure 6. Inputs to the algorithm is the set of violating $\mathcal{V}$ and conforming examples $\mathcal{C}$. It proceeds in the following steps:

Synthesize buggy pattern from $\mathcal{V}$: This is achieved by method `getConjunctiveSubRule()`. It first calls method `merge` to align examples in $\mathcal{V}$ using ILP (described in detail in Section 4.1 and 4.2). This constructs a UAPDG $\mathcal{A}$. Then, it identifies nodes in $\mathcal{A}$ that have a mapping to some node in *each* example. This is performed by method `project` and results in a UAPDG $\mathcal{A}_c$ (subscript c stands for common). Nodes in $\mathcal{A}_c$ are existentially quantified since they correspond to some node in each example in $\mathcal{V}$. This resultant UAPDG is translated to the buggy pattern $bp(\vec{x})$ using the translation described in Section 3.3.

Find subset $\mathcal{C}'$ of $\mathcal{C}$ which satisfy the buggy pattern This is accomplished by querying the SMT solver. If a conforming example satisfies the buggy pattern $bp(\vec{x})$, the solver also returns the model comprising the satisfying node assignments for $\vec{x}$. We map these nodes to the corresponding variables in $\vec{x}$ to get the set of VPDGs $\mathcal{C}'$. Note that the nodes in the VPDGs mapped to variables in $\vec{x}$ are frozen, since the examples satisfy the buggy pattern through these nodes.

Proc `synthesizeRule` $(\mathcal{V}, \mathcal{C})$

1 $\text{bp}(\vec{x})$, $\mathcal{V}^{\vec{x}}$:= `getConjunctiveSubRule`$(\mathcal{V}^{\emptyset})$.;
2 $\mathcal{C}'^{\vec{x}} := \{C^{\vec{x}} | C^{\vec{x}} \models \text{bp}(\vec{x}), C \in \mathcal{C}\}$.;
3 nbp := `synthesizeNBP`$(\mathcal{V}^{\vec{x}}, \mathcal{C}'^{\vec{x}})$.;
4 **let** nbp be of the form $\bigvee_i \exists \vec{y}.\ \text{nbp}_i(\vec{x}, \vec{y})$.;
5 $\text{R} := \exists \vec{x}.\text{bp}(\vec{x}) \wedge \neg \bigvee_i \exists \vec{y}.\ \text{nbp}_i(\vec{x}, \vec{y})$.;
6 **if** *all* $V \in \mathcal{V}$ *satisfy* R **then**
7 **return** R.
// R is consistent with $\mathcal{V}$ and $\mathcal{C}$
else
8 **return** FAIL.
// R is not consistent with $\mathcal{V}$ and $\mathcal{C}$
end

Procedure: Overall rule synthesis algorithm.

Proc `getConjunctiveSubRule` $(\mathcal{E}^{\vec{v_f}})$

1 $\mathcal{A}^{\vec{v_f}}$:= `merge`$(\mathcal{E}^{\vec{v_f}})$; $\mathcal{A}^{\vec{v_f},\vec{v_b}}$:= `assignVars`$(\mathcal{A}^{\vec{v_f}}, \vec{v_b})$.;
2 $\mathcal{A}_c^{\vec{v_f},\vec{v_b}}$:= `project`$_{\mathcal{E}^{\vec{v_f}}}(\mathcal{A}^{\vec{v_f},\vec{v_b}})$.;
3 $\phi(\vec{v_f}, \vec{v_b})$:= `getFormula`$(\mathcal{A}_c^{\vec{v_f},\vec{v_b}})$.;
4 $\mathcal{E}^{\vec{v_f},\vec{v_b}} := \{\,\text{project}_{E^{\vec{v_f}}}(\mathcal{A}^{\vec{v_f},\vec{v_b}}) \mid E^{\vec{v_f}} \in \mathcal{E}^{\vec{v_f}}\,\}$.;
5 **return** $\phi(\vec{v_f}, \vec{v_b})$, $\mathcal{E}^{\vec{v_f},\vec{v_b}}$.;

Procedure: Obtains a conjunctive subrule given a set of valuated examples.

Proc `synthesizeNBP` $(\mathcal{V}^{\vec{x}}, \mathcal{C}^{\vec{x}})$

1 **if** $|\ \mathcal{C}^{\vec{x}}\ | = 0$ **then return** *False*.;
2 partitionList := $[\mathcal{C}^{\vec{x}}]$; post := *True* .;
3 **while** *True* **do**
4 partition := partitionList.remove(0).;
5 **let** partition be $\{\mathcal{C}_1^{\vec{x}}, \cdots, \mathcal{C}_k^{\vec{x}}\}$.;
6 **for** $i \in 1 \cdots k$ **do**
7 $\text{nbp}_i(\vec{x}, \vec{y})$, _ := `getConjunctiveSubRule`$(\mathcal{C}_i^{\vec{x}})$.;
8 nbp:= nbp $\vee\ \exists \vec{y}.\ \text{nbp}_i(\vec{x}, \vec{y})$.;
end
9 **if** all $V^{\vec{x}} \in \mathcal{V}^{\vec{x}}$ satisfy $\neg$ nbp **then return** nbp.;
10 i := index s.t. $V^{\vec{x}} \in \mathcal{V}^{\vec{x}}$ satisfies nbp_i.;
11 **if** $|\ \mathcal{C}_i^{\vec{x}}\ | = 1$ **then return** FAIL.;
12 **for** $(\mathcal{C}_{i,l}^{\vec{x}}, \mathcal{C}_{i,r}^{\vec{x}})$ *in* `getCandidatePartitions`$(\mathcal{C}_i^{\vec{x}})$ **do**
13 partitionList.add(partition $\setminus \mathcal{C}_i^{\vec{x}} \cup \{\mathcal{C}_{i,l}^{\vec{x}}\} \cup \{\mathcal{C}_{i,r}^{\vec{x}}\}$)
end
end

Procedure: Synthesizes non-buggy pattern from a given set of valuated violating and conforming examples.

Proc `generateCandidatePartitions` $(\mathcal{C}^{\vec{x}})$

1 $\mathcal{A}^{\vec{x}}$:= `merge`$(\mathcal{C}^{\vec{x}})$.;
2 $\mathcal{A}_c^{\vec{x}}$:= `project`$_{\mathcal{C}^{\vec{x}}}(\mathcal{A}^{\vec{x}})$.;
3 candidateNodes := {n | n is action node, n $\notin \mathbb{N}(\mathcal{A}_c^{\vec{x}})$, n $\in \mathcal{N}_{\mathcal{A}^{\vec{x}}}(u)$, u $\in \mathbb{N}(\mathcal{A}_c^{\vec{x}})$}.
// $\mathcal{N}_{\mathcal{A}}$ is neighbor relation in $\mathcal{A}$;
// $\mathbb{N}(\mathcal{A})$ is the set of nodes for $\mathcal{A}$;
4 H(n) := `computeEntropy`(n) **for** all n $\in$ candidateNodes.;
5 **for** $n \in$ *candidateNodes s.t.* $H(n) < \min_n H(n) + \delta$ **do**
6 partitionList += $(\mathcal{C}_n^{\vec{x}}, \mathcal{C}_{\neg n}^{\vec{x}})$.;
end
7 **return** partitionList.;

Procedure: Obtains candidate partitions for a given set of valuated conforming examples (see Section 4.3).

Fig. 6. RhoSynth Algorithm

Synthesize non-buggy pattern from $\mathcal{C}'$: This is achieved by method `synthesizeNBP()`. Besides $\mathcal{C}'$, we also pass to this method violating examples $\mathcal{V}$ with a node assignment that satisfy the buggy pattern. Initially, all the conforming examples in the input constitute a single partition. For these examples, we use the ILP graph alignment to align unfrozen nodes and synthesize a non-buggy pattern (using method `getConjunctiveSubRule` as before). If the non-buggy pattern is unsatisfiable on all frozen violating examples, we return the synthesized non-buggy pattern. Otherwise, it implies that the non-buggy pattern synthesized from all examples in the partition is too general to reject violating examples, and we partition the set of conforming examples further. The partitioning algorithm is described in Section 4.3.

Example 4. Consider non-buggy pattern synthesis for the rule described in Section 2.2. Calling `getConjunctiveSubrule` on all conforming examples satisfying the buggy pattern returns a UAPDG $\mathcal{A}$ that is same as the synthesized buggy pattern, visualized in Figure 4(a). Violating examples

such as `code-befores` in Figure 1 satisfy $\mathcal{A}$. Therefore, conforming examples are partitioned and a separate non-buggy pattern is synthesized for each partition (UAPDGs in Figure 4(b-c)).

We next describe the different components of the rule synthesis algorithm in greater detail.

4.1 Maximal Graph Alignment using ILP

Graph alignment amongst the violating examples or conforming examples is one of the key steps in synthesizing the buggy or non-buggy patterns in a rule respectively. Since both the buggy and non-buggy patterns are existentially quantified, the synthesis problem is NP-hard [Haussler 1989]. Its hardness stems from different choices available for mapping existential variables to nodes in the example graphs. Alternatively, the synthesis problem can be seen as a search over graph alignments such that aligned nodes in different examples are mapped to the same existential variable. Note that different graph alignments will result in different synthesized formulas. Given multiple graphs, we iteratively align two graphs at a time[5]. We frame the problem of choosing a desirable graph alignment for a pair of example graphs as an ILP optimization problem. Since we want to synthesize formulas that precisely capture the code examples, we choose alignments that maximize the number of aligned nodes and edges.

The ILP objective to maximize graph alignment can also be viewed as minimizing the graph edit distance between code examples. Consequently, our reduction to ILP follows the binary linear programming formulation to compute the exact graph edit distance between two graphs [Lerouge et al. 2017]. The node and edge substitutions are considered cost 0 and node/edge additions and deletions are cost 1 operations. We include a detailed reduction to the ILP optimization problem in the supplementary appendix. Below, we describe some constraints on the ILP optimization that are specific to our application:

- We align action nodes only if they have the same label. So, a method call `foo` in one example cannot align with method `bar` in another example. However, data nodes that are, optionally, labeled with their types do not have this restriction. This allows us to align data nodes even when their types do not resolve, or when their types are related in the class hierarchy, for e.g., `InputStream` and `FileInputStream`.
- Two data nodes align only if they have at least one aligned incoming or outgoing edge. This restricts alignment of data nodes to only occurrences when they are defined or used by aligned action nodes.
- When synthesizing the non-buggy pattern, nodes frozen to the same variables in $\vec{x}$ are constrained to be aligned.

When synthesizing rules from code changes, we use the ILP formulation described in this section to perform a fine-grained graph differencing over PDGs in a code change (similar to the GumTree tree differencing algorithm [Falleri et al. 2014]). Graph differencing labels nodes in the PDGs with change tags: *unchanged*, *deleted* or *added*, depending upon whether the node is present in both code-before and code-after, in only code-before, and in only code-after. Now, while aligning the violating code-before's for synthesizing the buggy pattern, we require that the node alignment respect these change tags, i.e., nodes are aligned only if their change tags are the same. This constraint helps to focus the synthesized buggy pattern on the changed code. This constraint is removed when synthesizing the non-buggy pattern. This is because even variables $\vec{x}$ frozen in the conforming examples, by design, may not respect these change tags.

[5]This is an instance of a "multi-graph matching" problem. This requires optimization techniques beyond ILP, for e.g., alternating optimization [Yan et al. 2013]. We did not explore these solutions since pairwise graph alignment worked well for our application.

4.2 Synthesizing Conjunctive Subrules

In this section, we describe the procedure `getConjunctiveSubRule` to synthesize a conjunctive subrule from a given set of valuated examples. It has two main steps. First, UAPDGs that correspond to input valuated examples are iteratively `merged` pairwise to obtain a UAPDG $\mathcal{A}$. Second, we `project` $\mathcal{A}$ to nodes *present* in all input examples to obtain UAPDG $\mathcal{A}_c$. Given UAPDGs $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_c$ obtained by calling `merge` followed by a `project` over-approximates $\mathcal{A}_1 \vee \mathcal{A}_2$. This property ensures that $\mathcal{A}_c$ obtained after merging a set of input valuated examples is satisfied by all of them. We first describe the `merge` operation, followed by `project`.

Let $\mathcal{A}_1 = (N_1, E_1, Lat_1, M_{f1}, M_{b1})$ and $\mathcal{A}_2 = (N_2, E_2, Lat_2, M_{f2}, M_{b2})$ be two UAPDGs over the same set of free variables Var_f. Let $NM \subseteq N_1 \times N_2$ and $EM \subseteq E_1 \times E_2$ be the node and edge mappings that are obtained from the graph alignment step. We first extend these mappings to relations $NM^\epsilon \subseteq N_1 \cup \{\epsilon\} \times N_2 \cup \{\epsilon\}$ and $EM^\epsilon \subseteq E_1 \cup \{\epsilon\} \times E_2 \cup \{\epsilon\}$ such that $NM^\epsilon = NM \cup \{(n_1, \epsilon) \mid (n_1, \_) \notin NM\} \cup \{(\epsilon, n_2) \mid (\_, n_2) \notin NM\}$, and $EM^\epsilon = EM \cup \{(e_1, \epsilon) \mid (e_1, \_) \notin EM\} \cup \{(\epsilon, e_2) \mid (\_, e_2) \notin EM\}$. Then, $\texttt{merge}_{(NM^\epsilon, EM^\epsilon)}(\mathcal{A}_1, \mathcal{A}_2)$ returns a UAPDG $\mathcal{A} = (N, E, Lat, M_f, M_b)$ constructed as follows:

- $N = \{(n_1, n_2) | (n_1, n_2) \in NM^\epsilon\}$
- We first compute $E' = \{(n_1, n_2) \xrightarrow{(e_1, e_2)} (n_1', n_2') | (e_1, e_2) \in EM^\epsilon, e_i \neq \epsilon \Leftrightarrow n_i \xrightarrow{e_i} n_i'\}$. Since EM is an edge mapping obtained from the graph alignment step, it follows that $e_1 \neq \epsilon \wedge e_2 \neq \epsilon \Rightarrow label(e_1) = label(e_2)$. Further, from definition of EM^ϵ, it follows that $e_1 \neq \epsilon \vee e_2 \neq \epsilon$. Once E' is computed, $E = \{(n_1, n_2) \xrightarrow{e} (n_1', n_2') | (n_1, n_2) \xrightarrow{(e_1, e_2)} (n_1', n_2') \in E', e = \{label(e_1), label(e_2)\} \backslash \{\epsilon\}\}$.
- $Lat(n_1, n_2) = Lat_1(n_1) \sqcup Lat_2(n_2)$ where the join is applied point-wise to each node predicate at n_1 and n_2 (we assume that $Lat(\epsilon) = \bot$)
- $M_f(n_1, n_2) = x$, if $M_{f1}(n_1) = M_{f2}(n_2) = x \in Var_f$
- Nodes not mapped to free variables are mapped to distinct fresh variables– if $(n_1, n_2) \notin domain(M_f)$, then $M_b(n_1, n_2) = x_i$ for a distinct variable $x_i \in Var_b$.

Method `project` with respect to $\mathcal{A}_1$ and $\mathcal{A}_2$ projects the merged UAPDG $\mathcal{A}$ to $\mathcal{A}_c$ which has nodes (resp. edges) that map to nodes (resp. edges) in both $\mathcal{A}_1$ and $\mathcal{A}_2$, i.e., nodes in $\mathcal{A}_c$ are of the form (n_1, n_2) where $n_i \in N_i$ and edges are of the form (e_1, e_2) where $e_i \in E_i$.

Theorem 5. *For any node mapping $NM \subseteq N_1 \times N_2$ and edge mapping $EM \subseteq E_1 \times E_2$ that is obtained from aligning $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1 \vee \mathcal{A}_2 \Rightarrow \texttt{project}_{\{\mathcal{A}_1, \mathcal{A}_2\}}(\texttt{merge}_{(NM^\epsilon, EM^\epsilon)}(\mathcal{A}_1, \mathcal{A}_2))$.*

The proof outline for the above theorem is presented in the supplementary appendix. Besides, returning the formula for $\mathcal{A}_c$, method `getConjunctiveSubrule` also returns the set of valuated examples labeled with bound variables used for constructing $\mathcal{A}$. This is achieved by `projecting` $\mathcal{A}$ to nodes (resp. edges) that map to nodes (resp. edges) in each example. Note, to synthesize conjunctive subrules from a single example (possible in the case of non-buggy pattern synthesis), we heuristically include in the subrule all nodes at distance $d = 1$ from nodes bound to free variables $\vec{x}$.

4.3 Partitioning Conforming Examples

In this section, we describe the method `generateCandidatePartitions`. This method returns a list of low entropy partitions of the input examples $\mathcal{C}$, based on the clustering algorithm in Li et al. 2004. Entropy is widely used in machine learning, for e.g. in decision trees [Mitchell 1997]), and biases the partitioning towards sub-partitions that have homogeneous features. Let N be the set of nodes in $\mathcal{A}$ and N_c be the set of nodes in $\mathcal{A}_c$, where $\mathcal{A}$ and $\mathcal{A}_c$ are the UAPDGs obtained after calling `merge` and then calling `project` on examples in $\mathcal{C}$ respectively. For $n \in N$, let $\mathcal{C}_n = \{C_i\} \subseteq \mathcal{C}$ be the set of examples such that n maps to a node in C_i and let $\mathcal{C}_{\neg n} = \mathcal{C} \backslash \mathcal{C}_n$. Entropy of a partition

with respect to n is:
$H(\mathcal{C}, \mathcal{C}_n, \mathcal{C}_{\neg n}) = \frac{|\mathcal{C}_n|}{|\mathcal{C}|}.H(\mathcal{C}_n) + \frac{|\mathcal{C}_{\neg n}|}{|\mathcal{C}|}.H(\mathcal{C}_{\neg n})$,
$H(\mathcal{C}) = \sum_{n'} H_{n'}(\mathcal{C})$, where $H_{n'}(\mathcal{C}) = -p.\log p - (1-p).\log(1-p)$,
where $p = \mid \mathcal{C}_{n'} \mid / \mid \mathcal{C} \mid$. Nodes in N_c by definition are mapped to nodes in all input examples. Since code patterns in a rule are often localized, we consider partitions with respect to nodes in N that neighbor nodes in N_c. Further, we return the set of all partitions whose entropy is within a δ margin of the smallest entropy partition. Each of these partitions is explored in a BFS manner in method `synthesizeNBP`. The process stops at the first partition that synthesizes a non-buggy pattern that is unsatisfied by all violating examples.

4.4 Discussion about RhoSynth

Precision and Generalization: RhoSynth prefers precision over recall. For this reason, RhoSynth biases the buggy pattern to more specific formulas using an ILP-based maximal graph alignment, even if it may cause some amount of overfitting. In practice, we minimize overfitting by using diverse examples for synthesizing a rule, for instance, examples that are obtained from code changes that belong to different packages (refer to Section 6.1 for details). Furthermore, to prevent overfitting, RhoSynth biases non-buggy patterns to fewer disjuncts. This is accomplished by partitioning a group of correct examples only when the most-specific conjunctive non-buggy pattern synthesized from all correct examples satisfies a violating example. In this case, partitioning the correct examples and synthesizing a non-buggy disjunctive pattern for each partition becomes necessary. In addition, RhoSynth uses semi-lattices to express all node predicates. This helps generalization when merging groups of conforming and violating examples to synthesize the rule.

Soundness: We next show that RhoSynth is sound, i.e., given a set of violating and conforming examples, if RhoSynth synthesizes a rule R then R satisfies all violating examples and does not satisfy any conforming example. Formally,

Theorem 6 (Soundness of RhoSynth:). *Given a set of violating examples $\mathcal{V} = \{V_1, \cdots, V_m\}$ and conforming examples $\mathcal{C} = \{C_1, \cdots, C_n\}$, if the algorithm* `synthesizeRule` *successfully returns a rule R, then $V_i \models R$ and $C_i \not\models R$.*

We present a proof outline in the supplementary appendix. It is easy to argue Algorithm 6's soundness with respect to violating examples. To argue soundness with respect to conforming examples, we rely on the soundness of the `merge` algorithm (theorem 5). Using Theorem 5, we argue that all valuated conforming examples satisfy the synthesized non-buggy pattern. Since the rule negates the synthesized non-buggy pattern, none of them satisfy the synthesized rule.

Completeness: RhoSynth does not provide completeness guarantees. This is a design choice made purely based on practical experiments. Incomleteness may arise from incorrect graph alignments or incorrect partitioning of conforming examples when synthesizing non-buggy patterns. However, in our experiments, we do not encounter cases where either of these lead to rule synthesis failures or imprecise solutions.

5 IMPLEMENTATION DETAILS

PDG representation: We use RhoSynth to synthesize rules for Java. Our implementation uses MUDetect [Sven and Nguyen 2019] for representing Java source code as PDGs. MUDetect uses a static single assignment format. Further, we transform these PDGs using the following two program transformations to obtain program representations that are more conducive to rule synthesis:

(1) We abstract the label of relational operators such as $<, \leq, =, \neq$, etc. to a common label `rel_op`. This transformation is useful when synthesizing rules from examples that perform a similar check on a data value using different relational operators, for e.g., `error < 0` and `error == -1`. This transformation helps to express rules over such examples using a conjunctive formula over the `rel_op` label, instead of a disjunction over different relational operators.
(2) If a PDG has multiple calls to the same getter method on the same receiver, we transform it into a PDG that has a single getter call and the value returned by the getter is directly used at other call sites. So, we transform the code snippet `foo(url.getPort()); ... if (url.getPort() == 0) { ... }` to the PDG representation of the code snippet `v = url.getPort();foo(v); ... if (v == 0) { ... }` . Since RhoSynth, typically, does not have access to method declarations of called methods, we use heuristics on the name of the method to identify getter calls.

Lattices: To enable unification of node predicates at corresponding nodes in different PDGs, we lift all the node predicates to join semi-lattices in the following manner:

(1) Predicate *data-type(x)* and *declaring-type(x)* are lifted to a join semi-lattice that captures the longest common prefix (*lcp*) and longest common suffix (*lcs*) of the string representations of data types. Often, a Java class name shares a common prefix or a suffix with name of its superclass or the interfaces it extends. This semi-lattice is carefully designed to unify such related data types. If nodes n_1 and n_2 in different examples align to form node n where *data-type*(n_1) = `"BufferedInputStream"` and *data-type*(n_2) = `"FileInputStream"`, this semi-lattice helps generalize *data-type*(n) = `".*InputStream"`.
Formally, we represent a type t as a pair of strings (t, t). Join operation is defined as: $(tp, ts) \sqcup (rp, rs) = (lcp(tp, rp), lcs(ts, rs))$, where the top element in the semi-lattice is (ϵ, ϵ) for ϵ being an empty string.
(2) Predicates *label(x)*, *data-value(x)*, *num-para(x)*, *output-ignored(x)* are lifted to constant semi-lattices in their respective domains. The join operation is formally defined as:

$$p \sqcup q = \begin{cases} p & \text{if } p = q \\ \top & \text{otherwise} \end{cases}$$

As an example of this lattice in action during unification of predicates, if *num-para*(n_1) = 1 and *num-para*(n_2) = 2 and n_1 and n_2 align to form node n, then *num-para*(n) = $\top$.
(3) Predicate *trans-control-dep(x)* is lifted to a power-set semi-lattice with set intersection as the join operator. Formally, the join operator is defined as: $S_1 \sqcup S_2 = S_1 \cap S_2$. As an example of this lattice in action, if *trans-control-dep*(n_1) = {`If`, `Catch`} and *trans-control-dep*(n_2) = {`If`} and n_1 and n_2 align to form node n, then *trans-control-dep*(n) = {`If`}.

Solvers: We use the Xpress ILP solver for graph alignment and Z3 [De Moura and Bjørner 2008] for satisfiability solving. For running a rule on a method, we check if the rule is satisfied by the PDG representation of the method. We have implemented a few algorithmic simplifications that are used to simplify the satisfiability query, for e.g., we can determine that a rule, which asserts existence of a method call `foo`, will not be satisfied by a method that does not call `foo`. Due to such simplifications, >99% of the satisfiability queries can be discharged without calling Z3.

6 EVALUATION

To understand the effectiveness of RhoSynth, we address the following research questions.

RQ1: How effective is RhoSynth at synthesizing precise rules?
Section 6.2 shows that the precision of rules exceeds 75% in production.

RQ2: Are the synthesized rules "interesting"?
The rules synthesized cover diverse categories and are often discussed on various online forums as shown in Section 6.3.

RQ3: What is the effectiveness of iterative rule refinement in improving the precision?
As we show in Section 6.4, rule refinement improves the precision of rules, by as much as 68% in some cases.

RQ4: How does RhoSynth compare against state-of-the-art program synthesis approaches? The baselines used are ProSynth [Raghothaman et al. 2019] and anti-unification based synthesis over ASTs.
The baselines often fail to synthesize rules (Section 6.5).

RQ5: How does ILP-based graph alignment compare against various code alignment baselines? The baselines include GumTree [Falleri et al. 2014] and alignment algorithms based on brute-force enumeration and approximate polynomial-time graph matching.
Baseline algorithms do not perform well for rule synthesis and rule refinement applications (Section 6.6).

6.1 Experimental Methodology and Setup

Code Change Examples: We obtain code changes from 27,752 Java GitHub packages that we select based on their license, i.e., Apache or MIT-license, and star rating. The code changes are grouped by a clustering algorithm[6], which is tuned to output clusters that have a high homogeneity score. This ensures that all code-changes within the same cluster correspond to the same rule. We tag each code change with the relative file path, the name and the line number of the method containing the change. We use this triple as a criterion to deduplicate code-changes within a cluster. Further, we filter clusters containing less than three examples or having examples from less than two repositories or two commit ids. This ensures that the examples in the clusters are diverse and are not specific to a particular package. We then choose clusters containing popular APIs belonging to popular frameworks such as Java.util, Android, Apache, Guava, etc. The popularity is determined based on frequency of occurrence in the corpus. This results in 1397 clusters. We pick 300 clusters randomly from this set. We manually examined these clusters and select those clusters whose underlying APIs and code constructs find relevant results on a Google search, which is evidence for a general applicability of the code changes in the cluster. For e.g., code-changes for the "use-guava-hashmap" rule involve the `HashMap constructor` and `Maps.newHashMapWithExpectedSize` API that have an associated Stack-Overflow post. We spend on average 5 minutes for manually examining each cluster. This examination resulted in 31 clusters which are then input to RhoSynth.

Iterative Rule Refinement: We run the synthesized rules on code corpus. We identify rules for iterative refinement based on the detections generated by them. For each rule, we first label 5 detections. If any of these detection is labeled as a false positive (FP), we shortlist the rule for refinement. For such rules, we label up to 10 additional detections. With every false positive, we resynthesize the rule using the original set of code changes and all the FPs encountered till then. We stop when adding the N^{th} FP does not change the synthesized rule, i.e., rule synthesized after the first N FPs is equivalent to the rule synthesized after $(N-1)$ FPs.

Production Deployment: We have deployed all these rules with Amazon CodeGuru Reviewer. Internally, within Amazon, these rules are run on the source code at the time of each code review. Detections generated by these rules on the code diff are provided as code review comments. Code authors can label these comments as part of code review workflow.

[6]Refer to the supplementary appendix for details about the clustering algorithm.

Evaluation Setting: We perform two types of evaluation: *offline evaluation* by software developers on sampled recommendations, and ratings and textual feedback obtained during *live code reviews* from code authors at Amazon.

In both cases, developers label recommendations as "Useful", "Not Useful" or "Not Sure". The offline evaluation involves 10 expert developers that does not include the paper authors. In live code reviews, recommendations are labeled by developers who raised the code review or, in other words, the code authors. The evaluation using live code reviews spanned 5 months. During this time, the synthesized rules generated 2844 detections in 2414 CRs authored by 1238 unique developers. Besides recommendations from the synthesized rules, developers optionally used other tools such as Checkstyle, FindBugs, SonarQube, FaceBook Infer and Amazon CodeGuru, in their development workflow. During live code review, developers also provide textual feedback in addition to labeling the recommendations.

Metrics: For both the evaluations, we report *precision* = *# Useful* / *# Total labels*, computed over all labeled detections.

Experimental Setup: We conducted all our experiments on a Mac OSX laptop with 2.4GHz Intel processor and 16Gb memory. Experiments with ProSynth [Raghothaman et al. 2019] were run in a Docker container with 8Gb memory launched from a Docker image shared by the authors.

6.2 Precision of RhoSynth

In our offline evaluation, 91% detections (107 out of 117) were labeled "Useful". During live code reviews, we received developer feedback on detections from 25 out of the 31 rules. 75.8% of these labeled detections (273 out of 360) were categorized as "Useful" by the developers[7]. Some of the textual feedback from live code reviews are: `"Interesting, that is helpful"`, `"Will add this check."`, and `"we have a separate task to look into the custom polling solution https://XX"`. A high developer acceptance during code reviews shows that RhoSynth is capable of synthesizing code quality rules which are effective in the real-world.

Table 2 provides more information about these synthesized rules. Most of these rules are synthesized from few code changes. The average number of nodes in PDGs used for synthesizing these rules range from 5 to 367, with an average of 48 nodes. The synthesis time for these rules range from 30*ms* to 13*s*, with an average of 1.5*s*. Rules that only consist of a buggy pattern correspond to a code anti-pattern. 9 rules consist of both a buggy and non-buggy pattern. Of these, 4 rules have a disjunctive non-buggy pattern. These disjuncts correspond to different ways of writing code that is correct with respect to the rule. One example of such a rule is `check-movetofirst`, described in Section 2.2. Another example is `executor-graceful-shutdown` that intuitively checks if `ExecutorService.shutdownNow()` call is accompanied by a graceful wait implemented by calling `ExecutorService.awaitTermination()` or `ExecutorService.invokeAll()`. We include visualizations for few synthesized rules in the supplementary appendix.

6.3 Examples of RhoSynth Rules

We describe all the rules synthesized by RhoSynth in Table 1. A large number of these rules are supported by code documentation or discussions on online forums. They cover a wide range of code quality issues and recommendation categories:

- performance: e.g., use-parcelable, use-guava-hashmap
- concurrency: e.g., conc-hashmap-put, countdown-latch-await

[7]Most rules that did not receive developer feedback during live code reviews involved detecting code context with deprecated or legacy APIs. These rules trigger rarely during a code review and are thus more appropriate for a code scanning tool. All these rules were validated during offline evaluation.

- use of deprecated or legacy APIs: e.g., deprecated-base64, upgrade-enumerator
- bugs: e.g., check-movetofirst, start-activity
- code modernization: e.g., view-binding, layout-inflater
- code simplification: e.g., deserialize-json-array, use-collectors-joining
- debuggability: e.g., countdownlatch-await, exception-invoke

Table 1. The synthesized rules along with pointers to supporting blog-posts and code documentation.

S. No.	Rule name	API	Description of the synthesized rules
1	check-actionbar	Android getSupport-ActionBar	The method 'getSupportActionBar' returns 'null' if the Android activity does not have an action bar. One must null-check the value returned by 'getSupportActionBar', if the action bar is not explicitly set by a 'setSupportActionBar' call.
2	check-await-termination	Executor-Service.await-Termination	One must check the return value of 'awaitTermination()' to determine if the operation timed out while waiting for other threads to stop execution, following a shutdown request. Alternatively, one can check the same by calling'ExecutorService.isTerminated'.
3	check-create-newfile	File. create-NewFile	'createNewFile' returns False if a file already exists. One must check for 'File.exists' or check the value returned by 'createNewFile'. Without this check, one might silently overwrite an existing file leading to a data loss.
4	check-file-rename	FileSystem. rename	One must check if the Hdfs 'rename' operation succeeded and handle failures otherwise. This check can be performed by checking the value returned by the 'rename' call. Silent failures can lead to errors that are harder to debug.
5	check-inputstream-skip	Input-Stream. skip	'InputStream.skip' returns the number of bytes skipped. One must check the value returned by the 'skip' call to handle the case when fewer than the expected number of bytes are skipped.
6	check-mkdirs	File. mkdirs	One must check if the 'mkdirs' operation succeeded and handle failures otherwise. This check can be performed by checking the value returned by the 'mkdirs' call. Silent failures can lead to errors that are harder to debug.
7	check-movetofirst	Cursor. moveToFirst	One must check if the result set returned by a database query is empty. This can be performed by checking the value returned by 'Cursor.moveToFirst' or 'Cursor.isAfterLast', or checking if 'Cursor.getCount() > 0'. Without this check, the app can crash if subsequent operations are called on the cursor.
8	check-resultset-next	ResultSet. next	One must check if 'ResultSet.next()' returns False. If it returns False, it implies that the cursor is positioned after the last row and any subsequent calls to 'next()' will throw an exception.
9	conc-hashmap-put	Conc-urrentHash-Map.put	The rule detects code that calls 'ConcurrentHashMap.containsKey()', followed by a call to 'put()' if 'containsKey' returned 'False'. Since these operations are not atomic, the atomicity violation can lead to a data loss.
10	countdown-latch-await	Count-DownLatch. await	The 'await' method returns 'False' when the specified waiting time elapses while the thread is waiting for the latch to count down to zero. One must check the value returned by 'await' or check if 'CountDownLatch.getCount() > 0' to handle the case when the 'await' call had timed out.
11	create-list-from-map	Map. values	The rule detects code that converts the 'Collection' returned by 'Map.values()' into a Java 'Stream' and then collect it into a 'List'. This can be simplified by calling the 'ArrayList' constructor on the 'Collection' returned by 'values()'.
12	deprecated-base64	Spring Base64. encode	The 'Base64' class in the Spring framework's crypto library is deprecated. Instead of calling 'Base64.encode()', one must use 'encodeToString' declared in the 'java.util.Base64.Encoder' class. *URL:* *https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/codec/Base64.html*
13	deprecated-injectview	ButterKnife. inject	ButterKnife InjectView was deprecated in version 7. One must replace 'ButterKnife.inject(...)' with 'ButterKnife.bind()' or use Android's View Binding. *URL:* *https://github.com/JakeWharton/butterknife/blob/master/CHANGELOG.md#version-700-2015-06-27*
14	deprecated-mapping-exception	Jackson mapping-Exception	Method 'mappingException()' was deprecated in version 2.8 of the jackson-databind library. Instead use 'handleUnexpectedToken()'. *URL:* *https://fasterxml.github.io/jackson-databind/javadoc/2.8/com/fasterxml/jackson/databind/DeserializationContext.html#mappingException(java.lang.Class)*
15	deserialize-json-array	Gson. fromJson	The rule detects code that deserializes a list of JSON items by iterating in a loop. Instead, one can directly deserialize into a list by specifying the correct parameterized type using the 'TypeToken' class. *URL:* *https://github.com/google/gson/blob/master/UserGuide.md#TOC-Serializing-and-Deserializing-Generic-Types*
16	exception-invoke	Method. invoke	Whenever 'Method.invoke' is called, one must explicitly handle the 'Invocation-TargetException'. Since the actual underlying exception is the cause of 'Invocation-TargetException', it is desirable to call 'Throwable.getCause()' or 'getTargetException()' in the catch handler to access more information about the underlying exception.

17	executor-graceful-shutdown	ExecutorService.shutdownNow	One must shutdown an ExecutorService gracefully by first calling 'shutdown' to reject any incoming tasks, waiting a while for the existing tasks to terminate by calling 'awaitTermination', and then calling 'shutdownNow' to cancel lingering tasks. This is not required when the code calls 'ExecutorService.invokeAll' that waits till all the tasks complete. *URL: https://stackoverflow.com/questions/51819342*
18	layout-inflater	Android TextView constructor	One must inflate views using the 'LayoutInflater' instead of creating 'TextView''s programmatically in code. Especially, if the layout is complex, it is much easier to define it in XML and inflate it, rather than creating it all in code. *URL: https://developer.android.com/reference/android/view/LayoutInflater*
19	read-parcelable	Parcel.readValue	If one knows the specific type of the read object, one must use 'readParcelable' instead of calling 'readValue' followed by an explicit type-cast. Using 'readParcelable' will not require the type-cast operation.
20	replace-long-constructor	Long constructor	Instead of constructing a new 'Long' object, one must use 'Long.valueOf()' as this method yields better space and time performance by caching frequently requested values. *URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Long.html#valueOf-long-*
21	start-activity	Android startActivity	When launching an Android activity with 'startActivity(Intent, ...)', one must check 'Intent.resolveActivity(..)' for null. This checks if there exists an app on the device that can receive the implicit intent and launch the activity. Otherwise, the app will crash when 'startActivity' is called. This is not required when the activity is part of the same app, or when 'startActivity' is called within a try-catch block. *URL: https://developer.android.com/guide/components/intents-common*
22	upgrade-enumerator	Apache Enumerator constructor	The rule detects code that constructs an object of the deprecated 'Enumerator' class. One must use 'Collections.enumeration' instead. *URL: https://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/catalina/util/Enumerator.html*
23	upgrade-http-client	HttpClient.execute-Method	The rule detects code that calls 'HttpClient.executeMethod'. This method has been replaced in the HttpClient library version 4 with method 'execute'. One must upgrade to the latest version of the HttpClient library. *URL: https://stackoverflow.com/questions/40795037*
24	use-collectors-joining	Stream.collect	The rule detects code that collects all the items into a 'Collection' by calling 'Stream.collect()' and then joins them into a delimited String by 'join()'. Instead, one can use 'Collectors.joining(CharSequence)' to directly create a delimited String.
25	use-file-read-utility	HttpURL-Connection getInput-Stream	The rule detects code that creates a Reader from a Connection's input stream and reads its content by iterating in a loop. Instead, one can directly use a utility function, such as 'IOUtils.toString', to read the input stream. This makes the code more readable and also prevents any chances of resource leaks.
26	use-fs-is-dir	FileSystem.isDirectory	'FileSystem.isDirectory()' has been deprecated. One must use 'FileSystem.getFileStatus().isDir()' instead. *URL: https://hadoop.apache.org/docs/current/api/org/apache/hadoop/fs/FileSystem.html#isDirectory-org.apache.hadoop.fs.Path-*
27	use-guava-hashmap	HashMap constructor	The rule detects code that constructs a HashMap of a given size and then immediately adds all the keys into the map by iterating in a loop. The HashMap constructor uses a default load factor of 0.75, which means that the hash table will be rehashed after 75% of all keys have been added to the table. One must use Guava's 'newHashMapWithExpectedSize()' as that will not result in a rehash. *URL: https://stackoverflow.com/questions/30220820*
28	use-parcelable	Android getSerial-izableExtra	One must use 'Parcelable' instead of 'Serializable' to pass data between different components in Android, as the former is more performant. *URL: https://skoric.svbtle.com/serializable-vs-parcelable*
29	use-remove-if	Iterator.remove	The rule detects code that conditionally removes values from a hashmap by iterating over all the map entries in a loop. One can instead use 'Collection.removeIf'. This is also more efficient than removing values by iterating over all entries.
30	view-binding	Android findViewById	One must enable view binding in their module instead of calling 'findViewById()'. View binding provides null-safety and type-safety at compile time. *URL: https://developer.android.com/topic/libraries/view-binding#java*
31	wrapping-exception	Invocation-TargetException.getCause	The rule detects code that wraps 'InvocationTargetException.getCause()' into an unchecked exception. One must check if the 'Throwable' returned by 'getCause()' can be itself type-cast into an unchecked exception. A new unchecked exception must be constructed only if the type-casting operation is not successful.

6.4 Effectiveness of Iterative Rule Refinement

We iteratively refine 8 rules by providing additional non-buggy code examples. These examples correspond to variations in correct code that are not present in the code changes. As can be seen from Table 3, few additional examples are sufficient to refine the rule. All the 8 rules show precision

Table 2. Information about the synthesized rules. # new egs. are additional examples used for rule refinement. BP and NBP correspond to the buggy and (possibly disjunctive) non-buggy pattern. For both the buggy and non-buggy patterns we tabulate the number of nodes and the number of predicates. An entry $(i_1, i_2, \cdots)$ in the table indicates a non-buggy pattern whose disjuncts have respectively i_1, i_2, $\cdots$ nodes or predicates.

Rule name	# code changes	# new egs.	BP # nodes	BP # predicates	NBP # nodes	NBP # predicates
check-file-rename	6	-	4	9	-	-
check-inputstream-skip	5	-	3	6	-	-
check-mkdirs	10	-	1	4	-	-
check-resultset-next	4	-	8	27	-	-
conc-hashmap-put	3	-	9	24	-	-
create-list-from-map	3	-	8	20	-	-
deprecated-base64	4	-	4	10	-	-
deprecated-injectview	16	-	6	11	-	-
deprecated-mapping-exception	5	-	8	20	-	-
deserialize-json-array	3	-	10	26	-	-
layout-inflater	3	-	5	12	-	-
read-parcelable	7	-	6	14	-	-
replace-long-constructor	4	-	4	9	-	-
upgrade-http-client	3	-	7	16	-	-
upgrade-enumerator	6	-	6	16	-	-
use-collectors-joining	3	-	12	27	-	-
use-file-read-utility	3	-	15	40	-	-
use-fs-is-dir	4	-	4	8	-	-
use-guava-hashmap	3	-	20	59	-	-
use-parcelable	3	-	4	8	-	-
use-remove-if	3	-	11	30	-	-
view-binding	3	-	5	11	-	-
wrapping-exception	7	-	6	16	8	4
Rules Iteratively Refined						
check-actionbar	3	3	10	25	(14, 14)	(10, 9)
check-await-termination	5	2	8	21	12	12
check-createnewfile	3	3	4	10	6	6
check-movetofirst	4	9	2	5	(6, 8)	(11, 14)
countdownlatch-await	11	3	4	8	9	15
exception-invoke	4	3	3	5	6	8
executor-graceful-shutdown	4	4	2	5	(5, 10)	(6, 27)
start-activity	5	5	3	7	(5, 6, 11)	(7, 7, 20)

improvements and the (macro) average precision increases from 58% to 97% based on refinement. We use labels from offline evaluation for estimating the precision [8]. In 4 of the 8 rules, the refinement occurs by synthesizing disjunctive non-buggy patterns.

6.5 Comparison with Baselines

We compare RhoSynth against ProSynth [Raghothaman et al. 2019] and AST anti-unification based approaches for synthesizing rules without the rule refinement step. Getafix [Bader et al. 2019] and Revisar [Rolim et al. 2018] are representatives for synthesizing AST transformations via anti-unification of tree patterns. A direct comparison against Getafix and Revisar is unfortunately not feasible as the former is not publicly available and the latter does not support Java. Hence, we compare against our own implementation of an anti-unification algorithm over ASTs. The results are described in Table 4.

[8]We group all detections by the rule version, and estimate the overall precision based on 10 labels for each group.

Table 3. Results for iterative rule refinement. NBP size is the number of nodes in each disjunct of the synthesized non-buggy pattern; Prec. is the precision based on offline evaluation.

Rule name	# code changes	Before rule refinement		# new egs.	After rule refinement	
		NBP size	Prec.		NBP size	Prec.
check-actionbar	3	14	32%	3	(14, 14)	100%
check-await-termination	5	-	79%	2	12	100%
check-create newfile	3	-	42%	3	6	100%
check-movetofirst	4	-	40%	9	(6, 8)	100%
countdown latch-await	11	-	80%	3	9	100%
exception-invoke	4	9	76%	3	6	100%
executor-graceful-shutdown	4	9	95%	4	(5, 10)	100%
start-activity	5	9	21%	5	(5, 6, 11)	80%
Macro Avg.			58%			97%

ProSynth: ProSynth is a general algorithm to synthesize Datalog programs. It is not particularly tailored to the rule synthesis problem, which is the focus of our work, for two main reasons. First, a Datalog program can only express rules that are purely existentially quantified formulas without a non-buggy pattern [Ajtai and Gurevich 1994]. ProSynth is thus not able to synthesize rules that have a non-buggy pattern. It either times out or returns "Problem unsatisfiable" when synthesizing such rules.

Second, in cases when ProSynth is able to synthesize a rule, the synthesized rule often misses the required code context, i.e., the API of interest or constructs of interest. Synthesizing a rule from few examples is in most cases an underspecified problem. While RhoSynth biases the synthesis of buggy patterns towards larger code context, ProSynth uses a SAT solver based enumeration to synthesize a buggy pattern without such an inductive bias. As a result, rules synthesized by ProSynth can miss the required code context as long as they can differentiate between the few positive and negative code examples [9].

Experiment Methodology: We encode the buggy pattern synthesis problem in ProSynth as follows. For each PDG G, we introduce an auxiliary node v_G and an auxiliary binary relation BR that holds between every node in G and v_G. The buggy pattern bp is a unary output relation $bp(v_{V_k}) = \texttt{True}$ for all violating graphs V_k and `False` otherwise. We also extract all atomic node and edge predicates from the input PDGs and assert them as input tuples. When synthesizing a buggy pattern over i existential variables:

$$bp(v) :- P_i(x_1, x_2, \cdots, x_i), BR(x_1, v), \cdots, BR(x_i, v)$$

where P_i is an i-ary predicate defined recursively using P_{i-1} and all combinations of the i^{th} node x_i being connected to a subset of first $i-1$ nodes using different edge relations. In the base case, $P_1(x_1)$ is a conjunction of any number of atomic node predicates at x_1. We start with $i = 1$ and

[9] As an example, the code context for `conc-hashmap-put` rule must check the result of `ConcurrentHashMap.containsKey` followed by a call to `put`, on the same hash map, if `containsKey` returned `False`. The rule synthesized by ProSynth just checks for an existence of a `containsKey` call and misses the remaining code context.

Table 4. Comparison with baselines. TO means timeout at 15m; MC means missing code context; UnSAT means unsatisfiable problem.

Rule name	ProSynth	AST anti-unification	RhoSynth (with poly. graph matching)
deprecated-injectview			
deprecated-mapping-exception			
layout-inflater			
replace-long-constructor			
use-parcelable			✗
view-binding			✗
upgrade-enumerator			
check-actionbar	✗, TO		
check-createnewfile	✗, TO		
check-file-rename	✗, TO		
check-movetofirst	✗, TO		
read-parcelable	✗, MC		
upgrade-http-client	✗, MC		
use-fs-is-dir	✗, MC		
wrapping-exception	✗, TO		
check-inputstream-skip	✗, TO		✗
conc-hashmap-put	✗, MC		✗
create-list-from-map	✗, MC		✗
deprecated-base64	✗, MC		✗
check-await-termination	✗, TO	✗	
check-mkdirs	✗, TO	✗	
check-resultset-next	✗, TO	✗	
countdownlatch-await	✗, UnSAT	✗	
exception-invoke	✗, TO	✗	
start-activity	✗, TO	✗	
use-collectors-joining	✗, MC	✗	
use-file-read-utility	✗, MC	✗	
use-guava-hashmap	✗, MC	✗	
deserialize-json-array	✗, MC	✗	✗
executor-graceful-shutdown	✗, TO	✗	✗
use-remove-if	✗, MC	✗	✗

increase i. We report the results for the run with largest i that does not encounter a timeout (= $15m$). If ProSynth times out for $i = 1$, we report timeout (TO) in Table 4.

Results: ProSynth times out on 12 rules and returns "Problem unsatisfiable" for 1 rule. On a manual examination of the 18 rules it synthesized, we found that 11 rules do not contain the required code context. These rules would lead to false positives. Hence, we conclude that ProSynth is able to precisely synthesize 7 out of 31 rules.

Anti-unification over ASTs: Given two ASTs, we look at all combination of subtrees rooted at different nodes in the two ASTs. We pick the AST pattern, obtained on anti-unification of these subtrees, that has the largest size and contains the API of interest (column 3 in Table 1). We manually examine the AST patterns obtained on anti-unifying `code-before` and `code-after` examples. Anti-unification fails to capture the surrounding code context when input examples

```
1 boolean check = dir.mkdirs();
2 if (!check) { ··· }
3 ···
```

```
1 ···
2 if (!dir.mkdirs()) {···}
3 ···
```

Fig. 7. Two snippets of `code-after` input to the `check-mkdirs` rule

Table 5. Comparison of ILP based graph alignment with GumTree on aligning unpaired code examples for rule synthesis. Node mappings are only reported for those partitions of conforming examples that contain at least two new unpaired examples. We compare alignment over just action nodes since they have a 1-1 map between ASTs and PDGs.

Iteratively refined rule	NBP disjunct	# new unpaired egs.	# action nodes in rule NBP	# Node mappings using ILP graph-alignment	#Node mappings using GumTree	NBP synthesis succeeds with GumTree?
check-actionbar	nbp_1	-	6	-	-	
	nbp_2	3	6	18/18 = 100%	18/18 = 100%	✓
check-await-termination	nbp_1	2	6	6/6 = 100%	4/6 = 66%	✓
check-createnewfile	nbp_1	3	4	12/12 = 100%	7/12 = 58%	✗
check-movetofirst	nbp_1	6	3	45/45 = 100%	41/45 = 91%	✗
	nbp_2	3	4	12/12 = 100%	10/12= 83%	✓
countdownlatch-await	nbp_1	3	5	15/15 = 100%	7/15 = 47%	✗
exception-invoke	nbp_1	3	3	9/9 = 100%	3/9 = 33%	✗
executor-graceful-shutdown	nbp_1	1	2	-	-	
	nbp_2	3	6	18/18 = 100%	9/18 = 50%	✗
start-activity	nbp_1	3	2	6/6 = 100%	6/6 = 100%	✓
	nbp_2	1	3	-	-	
	nbp_3	-	5	-	-	

have syntactic variations in their AST representation. As an example, consider code snippets in Figure 7 that are input to the `check-mkdirs` rule. The code snippets have the same PDG representation. But their AST representation differs because of an explicit variable assignment in the first snippet. Aligning the ASTs at the `mkdirs()` call and anti-unifying them loses the crucial code context of checking the value returned by `mkdirs()` in an if-condition.

Results: We observe that anti-unification similarly misses the required code context in total 12/31 rules. The synthesized AST patterns in these cases would lead to false positives. This approach is thus able to precisely synthesize at most 19 rules. This experiment shows the limitation of AST anti-unification based approaches for synthesizing rules.

6.6 Comparing ILP-Based Graph Alignment against Baselines

We compare the ILP graph-alignment algorithm that forms the basis of RhoSynth against three baseline code alignment algorithms: GumTree [Falleri et al. 2014], brute-force enumeration, and an approximate, polynomial-time, graph matching algorithm.

GumTree: GumTree [Falleri et al. 2014] is a popular choice for performing AST differencing. We compare ILP-based alignment with GumTree on *aligning unpaired code examples for rule synthesis*. We observe that GumTree is effective when aligning code changes in which the two snippets overlap a lot. If the examples are "unpaired" (or have a small overlap), GumTree fails in some cases to precisely align them. We consider 9 scenarios covering all 8 iteratively refined rules where a conjunctive non-buggy pattern is synthesized from a partition of unpaired conforming code examples.

Results: In Table 5, we tabulate the number of nodes mappings established by the two alignment algorithms for every pair of code example in the partition. The number of nodes mapped by GumTree range from 33% to 91% in the failed scenarios. ILP maps all the nodes.

When we use node mappings established by the GumTree algorithm to `merge` these examples, we succeed in synthesizing the desired non-buggy pattern in only 4/9 scenarios. In the remaining scenarios, GumTree fails to map some nodes that form the required context for the non-buggy pattern, and are hence omitted on subsequent merge operations. These rules would lead to false

positives. The ILP-formulation leads to successful synthesis of the specific non-buggy pattern in all 9 scenarios.

Brute-force enumeration: We compare RhoSynth's graph alignment algorithm against a brute-force enumeration of graph alignments, for synthesizing buggy patterns. Given V existential variables, there are $O(N^{V.K})$ different graph alignments, where N is the number of nodes in the input examples and K is the number of examples. Here, the term $N^{V.K}$ corresponds to selecting V nodes in each of the K graphs. To compare against a brute-force enumeration strategy, we have implemented our own baseline that recursively enumerates all possible graph alignments in-memory. The algorithm stops exploring a partial graph alignment if the choice of already-selected nodes makes it infeasible, for e.g., when the graph alignment maps action nodes with different labels. We set V to the number of nodes in the buggy patterns synthesized by RhoSynth (column 4 in Table 2). The enumeration algorithm returns the alignment with the maximal matching measured by the number of predicates in the synthesized buggy pattern.

Results: Brute-force enumeration of graph alignments works for only 1/31 rules– specifically, the `executor-graceful-shutdown` rule. For the remaining, it times out on 15/31 rules (time out = 1 day) and runs out of memory on 15/31 rules (maximum allowed heap memory = 48Gb). In contrast the ILP graph alignment obtains a good graph alignment solution for all rules in at most a few seconds. This experiment shows that a brute-force enumeration of graph alignments is an infeasible approach for rule synthesis.

Approximate, polynomial-time, graph matching algorithm: We compare RhoSynth's graph alignment algorithm against an approximate, polynomial-time, graph matching baseline. Unlike RhoSynth that uses ILP for a graph-level optimization, in this baseline we match nodes only based on their local node features. Specifically, we hash each node in the graph to a key: the key for a data node is its type or the action node that defines the data value, and the key for an action node is the node label. Note that the key does not uniquely identify all nodes in a graph. For graph alignment, we incrementally match nodes in the given graphs in a pairwise manner. We first pair nodes that have the same key such that the key uniquely identifies both nodes in their respective graphs. If there are no such nodes, we pick any pair of nodes with the same key, pair them and iterate.

Results: Rule synthesis with polynomial graph matching is faster– the average synthesis time with this approach is 0.3*s* vs 1.5*s* when ILP graph alignment is used. However, this graph alignment approach is not optimal with respect to the specificity of the synthesized buggy and non-buggy patterns. Based on a manual investigation of the synthesized rules, 9/31 rules miss crucial code context in the buggy pattern with polynomial graph matching (see column 4 in Table 4). These rules generate 3× more detections, where most of the new detections are false positives. For e.g., the `conc-hashmap-put` rule with polynomial graph matching misses that the rule must flag code only when `ConcurrentHashMap.put()` call is within an if-statement that checks `containsKey` on the same map. Quantitatively, polynomial graph matching leads to 25% fewer predicates in the buggy pattern synthesized from the same code examples (average 12.5 predicates vs 16.4 predicates with ILP alignment). Our results show that using ILP graph alignment is a good choice for the rule synthesis application.

6.7 Discussion: Readability and Maintainability of Synthesized Rules

For production deployment, we automatically compile the synthesized rules into the Guru Query Language (GQL) [Mukherjee et al. 2022], which is a Java-based, domain-specific language used at Amazon for creating static code analysis rules. As an example, in Figure 8, we provide the synthesized GQL code for the `check-movetofirst` rule from Figure 4. At a high level, the GQL rule

```
protected CustomRule buildCustomRule() {
       return new CustomRule.Builder()
           .withDataByTypeFilter(true, "Cursor").as("x0")
           .withDataDependentsTransform(true, true, true)
           .withReceiverByIdFilter("x0")
           .withMethodCallFilter("moveToFirst")
           .withNumberOfArgumentsFilter(0)
           .withOutputIgnoredFilter().as("x1")
           .check()
           .withOneOf(true, true,
              b -> b
                  .withIdFilter("x0")
                  .withDataDependentsTransform(true, true, true)
                  .withReceiverByIdFilter("x0")
                  .withMethodCallFilter("isAfterLast")
                  .withNumberOfArgumentsFilter(0)
                  .withDefinitionTransform().as("y2")
                  .withDataDependentsTransform(true, true, true)
                  .withArgumentByIdFilter(0, "y5")
                  .withActionFilter("!")
                  .withDefinitionTransform()
                  .withDataByTypeFilter(true, "boolean"),
              b -> b
                  .withIdFilter("x0")
                  .withDataDependentsTransform(true, true, true)
                  .withReceiverByIdFilter("x0")
                  .withMethodCallFilter("getCount")
                  .withNumberOfArgumentsFilter(0)
                  .withDefinitionTransform().as("y2")
                  .withDataDependentsTransform(true, true, true)
                  .withArgumentByIdFilter(0, "y2")
                  .withActionFilter(true, "==", ">", ">=", "<", "<=", "!=").as("y4")
                  .withDefinitionTransform()
                  .withUsersTransform()
                  .withControlFilter("IF")
                  .withControlDependentsTransform()
                  .withIdFilter("x1")
                  .reset()
                  .withDataByTypeFilter(true, "int").as("y3")
                  .withDataDependentsTransform(true, true, true)
                  .withArgumentByIdFilter(1, "y3")
                  .withIdFilter("y4")
           )
           .build();
}
```

Fig. 8. Synthesized check-movetofirst rule in the Guru Query Language (GQL) used at Amazon.

incrementally asserts and checks, in the input code example, all the node and edge predicates that comprise the buggy pattern and non-buggy pattern. All assertions before `check()` comprise the buggy pattern and all assertions after `check()` comprise the non-buggy pattern. The construct `withOneOf()` introduces a disjunction in the non-buggy pattern. The details of the GQL syntax and semantics are described in Mukherjee et al. 2022. As a simple illustration of the checks asserted in the synthesized GQL rule, the sequence of calls in lines 3 – 6 in Figure 8 first assert the presence of a PDG node with a data type `Cursor`; the node is then tagged `"x0"`; then, the query checks for a data-dependent node that corresponds to a method call `moveToFirst` with `"x0"` as its receiver.

Figure 9 shows the GQL code for the `check-movetofirst` rule written by an expert GQL developer. Since, RhoSynth uses a core subset of GQL APIs to express the synthesized rules, an expert GQL developer may use APIs outside this subset to express the same rule more succinctly. Even though the synthesized rule may be slightly longer than the manually written rule, it is fairly readable. Within Amazon, these rules are code-reviewed and maintained by developers like any other piece of Java code. We have observed that visualizing the synthesized UAPDGs as graphs, for e.g. the UAPDGs in Figure 4(a)-(c)), helps read and understand the synthesized rules. Hence, along with the synthesized Java code, we also share these UAPDGs for reviewing and understanding the rules.

```
1 protected CustomRule buildCustomRule() {
2     return new CustomRule.Builder()
3         .withMethodCallFilter("moveToFirst")
4         .withNumberOfArgumentsFilter(0)
5         .withOutputIgnoredFilter().as("x1")
6         .withReceiverTransform()
7         .withDataByTypeFilter(true, "Cursor")
8         .as("x0")
9         .check()
10         .withOneOf(true, true,
11           b -> b
12               .withDataByTypeFilter(true, "boolean")
13               .withDefinedByTransform()
14               .withActionFilter("!")
15               .withArgumentTransform(0)
16               .withDefinedByTransform()
17               .withMethodCallFilter("isAfterLast")
18               .withNumberOfArgumentsFilter(0)
19               .withReceiverByIdFilter("x0")
20           ,
21           b -> b
22              .as("y5")
23              .withDefinedByTransform()
24              .withActionFilter(true, "==", ">",
                          ">=", "<", "<=", "!=")
25              .withArgumentByTypeFilter(1, "int")
26              .withArgumentTransform(0)
27              .withDefinedByTransform()
28              .withMethodCallFilter("getCount")
29              .withNumberOfArgumentsFilter(0)
30              .withReceiverByIdFilter("x0")
31              .reset()
32              .withId("y5")
33              .withUsersTransform()
34              .withControlFilter("IF_STATEMENT")
35              .withControlDependentsTransform()
36              .withIdFilter("x1")
37       )
38       .build();
39 }
```

Fig. 9. `check-movetofirst` rule written by an expert developer in GQL.

7 RELATED WORK

Synthesis algorithms for program repair and bug detection: Our work is most closely related to research on synthesis algorithms for automated program repair from code changes [Bader et al. 2019; Bavishi et al. 2019; Meng et al. 2013; Miltner et al. 2019; Rolim et al. 2017, 2018]. We can categorize these works based on their techniques. The first category consists of Phoenix [Bavishi et al. 2019] and instantations of the PROSE framework [Polozov and Gulwani 2015] in Refazer [Rolim et al. 2017] and BluePencil [Miltner et al. 2019]. These algorithms search for a program transformation, which explain the set of provided concrete edits, within a DSL. The second category includes Revisar [Rolim et al. 2018], Getafix [Bader et al. 2019] and

LASE [Meng et al. 2013] that are based on generalization algorithms over ASTs, using anti-unification or maximum common embedded subtree extraction. Both the above categories of work either rely on an accompanying static analyzer for bug localization or synthesize program transformation patterns that are specific to a given package or domain. Bug localization is harder than repair at a given location [Allamanis et al. 2021]. Our approach does not rely on existing static analyzers for bug localization and is able to precisely synthesize new rules. These rules are also applicable to code variations that are present in different packages. Additionally, we observe that correct code may have variations that is not present in code changes. We present an approach to refine rules using additional examples containing such code variations. In comparison, most of the above mentioned prior works rely on paired code changes and none of them support iterative refinement of rules.

DiffCode [Paletov et al. 2018] is an approach to infer rules from code changes geared towards Crypto APIs. Their main focus is on clustering algorithms for code changes. They do not automate the task of synthesizing a rule from a cluster of code changes. Program synthesis has also been recently applied to other code related applications such as API migration [Gao et al. 2021; Ni et al. 2021], synthesis of merge conflict resolutions [Pan et al. 2021] and interactive code search [Naik et al. 2021; Sivaraman et al. 2019]. While the interactive code search application has similarities with rule synthesis, there are differences in the technical approach followed in these prior works. Sporq [Naik et al. 2021] uses Datalog query synthesis and Alice [Sivaraman et al. 2019] uses inductive logic programming, as opposed to integer linear programming based graph alignment in RhoSynth. Besides these work also differ in the format of code patterns that a developer can search in their code base.

Datalog synthesis: There is a rich body of recent work on Datalog synthesis [Albarghouthi et al. 2017; Mendelson et al. 2021; Raghothaman et al. 2019; Si et al. 2018, 2019; Thakkar et al. 2021] and its application to code related tasks such as interactive code search [Naik et al. 2021]. Datalog programs can express existentially quantified buggy patterns [Ajtai and Gurevich 1994] but cannot express rules with a non-vacuous non-buggy pattern that introduce quantifier alternation. On the other hand, these algorithms can synthesize recursive predicates that we exclude in our approach. In Section 6, we provide an empirical comparison against ProSynth [Raghothaman et al. 2019] from this category.

Statistical approaches for program synthesis: Dinella et al. 2020; Tufano et al. 2018 are neural approaches for automatic bug-fix generation. These models are trained on a general bug-fix dataset, not necessarily fixes that correspond to rules, and have a comparatively lower accuracy. TFix [Berabi et al. 2021] improves upon them by fine-tuning the neural models on fixes that correspond to a known set of bug categories or rules. In doing so, it gives up the ability to generate fixes for new rules. Devlin et al. 2017b; Long and Rinard 2016 propose a hybrid approach where algorithmic techniques are used to generate candidate fixes and statistical models are used to rank them. Allamanis et al. 2018, 2021; Pradel and Sen 2018; Vasic et al. 2019 present neural models for detecting bugs caused by issues in variable naming and variable misuse. These approaches cannot be easily tailored to generate new detectors from a given set of labeled code examples.

In the domain of synthesizing String transformations, pioneered by FlashFill [Gulwani 2011], RobustFill [Devlin et al. 2017a] presents a neural approach for program synthesis as well as program induction. Hybrid approaches such as Gulwani and Jain 2017; Kalyan et al. 2018; Verbruggen et al. 2021 complement machine learning based induction with algorithmic techniques and present an interesting direction for exploration in the context of rule synthesis.

Statistical approaches for building bug-detectors include data-mining approaches such as PR-Miner [Li and Zhou 2005], APISan [Yun et al. 2016] and NAR-miner [Bian et al. 2018]. These

approaches mine popular programming patterns and flag deviants as bugs. Since these mined patterns are based on frequency, these approaches are not able to distinguish between incorrect code and infrequent code. Recent work on mining code patterns has achieved higher precision in finding bugs when the mined patterns are applied to the same code-base they are extracted from [Ahmadi et al. 2021]. Arbitrar [Li et al. 2021] improves upon APISan [Yun et al. 2016] by incorporating user-feedback through active learning algorithms. Both Arbitrar and APISan employ symbolic execution to extract semantic features of a program, as opposed to a lighter-weight static analysis in RhoSynth. Rules based on symbolic execution might be too expensive for a real-time code reviewing application.

Interactive program synthesis: Researchers have recently explored the question-selection problem [Ji et al. 2020] and other user-interaction models [Bavishi et al. 2021; Ferdowsifard et al. 2021; Le et al. 2017] in the context of interactive program synthesis. Rule refinement is also an instance of interactive program synthesis and applying these approaches to the domain of rule synthesis is an interesting research direction.

DSLs for expressing rules in static checkers: Most static checkers, such as SonarQube[10], PMD[11], Semmle and Semgrep, allow users to write their own custom rules. These rules are often written in a DSL such as ProgQuery [Rodriguez-Prieto et al. 2020] or CodeQL. On the same lines, GQL [Mukherjee et al. 2022] is a Java-based DSL used at Amazon for creating code analysis rules. We are not aware of any tool that synthesizes rules in these DSLs automatically from labeled code examples. In fact, a recent survey [Raychev 2021] identifies automating the rule creation process in static checkers as a largely unexplored area.

Synthesis frameworks: SyGus [Alur et al. 2013] and CEGIS [Solar Lezama 2008] are two popular synthesis frameworks in domains with logical specifications. Recently, Wang et al. [Wang et al. 2021] have proposed an approach that combines SyGus with decision tree learning for synthesizing specific static analyses that detect side-channel information leaks.

8 CONCLUSIONS

In this paper, we present RhoSynth, a new algorithm for synthesizing code-quality rules from labeled code examples. RhoSynth performs rule synthesis on graph representations of code and is based on a novel ILP based graph alignment algorithm. We validate our algorithm by synthesizing more than 30 rules that have been deployed in Amazon CodeGuru Reviewer. Our experimental results show that the rules synthesized by our approach have high precision (greater than 75% in internal live code reviews) which make them suitable for real-world applications. In the case of low-precision rules, we show that rule refinement can leverage additional examples to incrementally improve the rules. Interestingly, these synthesized rules are capable of enforcing several documented code-quality recommendations. Through comparisons with recent baselines, we show that current state-of-the-art program synthesis approaches are unable to synthesize most rules.

ACKNOWLEDGMENTS

We thank Rajdeep Mukherjee for providing the dataset comprising GitHub code change clusters that we used for synthesizing rules, and Omer Tripp and Ben Liblit for discussions and feedback. We also thank the reviewers for their insightful feedback which substantially improved the presentation of the paper.

[10] https://docs.sonarqube.org/latest/extend/adding-coding-rules/
[11] https://pmd.github.io/latest/pmd_userdocs_extending_writing_pmd_rules.html

REFERENCES

Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. 2021. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2025–2040. https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi

Miklós Ajtai and Yuri Gurevich. 1994. Datalog vs First-Order Logic. *J. Comput. Syst. Sci.* 49, 3 (1994), 562–588. https://doi.org/10.1016/S0022-0000(05)80071-6

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Principles and Practice of Constraint Programming*, J. Christopher Beck (Ed.). Springer International Publishing, Cham, 689–706. https://doi.org/10.1007/978-3-319-66158-2_44

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJOFETxR-

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. *CoRR* abs/2105.12787 (2021). arXiv:2105.12787 https://arxiv.org/abs/2105.12787

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 1–17. https://doi.org/10.1109/FMCAD.2013.6679385

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories*. Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

Rohan Bavishi, Caroline Lemieux, Koushik Sen, and Ion Stoica. 2021. Gauss: Program Synthesis by Reasoning over Graphs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 134 (oct 2021), 29 pages. https://doi.org/10.1145/3485511

Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

Kent Beck. 2002. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. https://proceedings.mlr.press/v139/berabi21a.html

Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-Miner: Discovering Negative Association Rules from Code for Bug Detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 411–422. https://doi.org/10.1145/3236024.3236032

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017a. RobustFill: Neural Program Learning under Noisy I/O. *CoRR* abs/1703.07469 (2017). arXiv:1703.07469 http://arxiv.org/abs/1703.07469

Jacob Devlin, Jonathan Uesato, Rishabh Singh, and Pushmeet Kohli. 2017b. Semantic Code Repair using Neuro-Symbolic Transformation Networks. *CoRR* abs/1710.11054 (2017). http://arxiv.org/abs/1710.11054

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 153 (oct 2021), 29 pages. https://doi.org/10.1145/3485530

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: Output-Oriented Program Synthesis for Combating Breaking Changes in Libraries. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 161 (oct 2021), 27 pages. https://doi.org/10.1145/3485538

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *SIGPLAN Not.* 46, 1 (Jan. 2011), 317–330. https://doi.org/10.1145/1925844.1926423

Sumit Gulwani and Prateek Jain. 2017. Programming by Examples: PL Meets ML. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10695)*, Bor-Yuh Evan Chang (Ed.). Springer, 3–20. https://doi.org/10.1007/978-3-319-71237-6_1

David Haussler. 1989. Learning Conjunctive Concepts in Structural Domains. *Mach. Learn.* 4 (1989), 7–40. https://doi.org/10.1007/BF00114802

Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1143–1158. https://doi.org/10.1145/3385412.3386025

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rywDjg-RW

Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M. Eskofier, and Michael Philippsen. 2016. Automatic Clustering of Code Changes. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2901739.2901749

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. *CoRR* abs/1703.03539 (2017). arXiv:1703.03539 http://arxiv.org/abs/1703.03539

Julien Lerouge, Zeina Abu-Aisheh, Romain Raveaux, Pierre Héroux, and Sébastien Adam. 2017. New binary linear programming formulation to compute the graph edit distance. *Pattern Recognition* 72 (Dec. 2017), 254 – 265. https://doi.org/10.1016/j.patcog.2017.07.029

Tao Li, Sheng Ma, and Mitsunori Ogihara. 2004. Entropy-based criterion in categorical clustering. In *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004 (ACM International Conference Proceeding Series, Vol. 69)*, Carla E. Brodley (Ed.). ACM. https://doi.org/10.1145/1015330.1015404

Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. ARBITRAR: User-Guided API Misuse Detection. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1400–1415. https://doi.org/10.1109/SP40001.2021.00090

Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 306–315. https://doi.org/10.1145/1081706.1081755

Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (jan 2016), 298–312. https://doi.org/10.1145/2914770.2837617

Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. 2021. GENSYNTH: Synthesizing Datalog Programs without Language Bias. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 6444–6453. https://ojs.aaai.org/index.php/AAAI/article/view/16799

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 502–511.

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 143 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360569

Tom M. Mitchell. 1997. *Machine Learning*. McGraw-Hill, New York.

Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. 2022. Static Analysis for AWS Best Practices in Python Code. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.14

Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. 2021. Sporq: An Interactive Environment for Exploring Code Using Query-by-Example. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3472749.3474737

Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Inês Lynce, Vasco M. Manquinho, Ruben Martins, and Claire Le Goues. 2021. SOAR: A Synthesis Approach for Data Science API Refactoring. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 112–124. https://doi.org/10.1109/ICSE43902.2021.00023

Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Inferring Crypto API Rules from Code Changes. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 450–464. https://doi.org/10.1145/3192366.3192403

Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu K. Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 785–796. https://doi.org/10.1109/ICSE43902.2021.00077

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2019. Provenance-Guided Synthesis of Datalog Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 62 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371130

Veselin Raychev. 2021. Learning to Find Bugs and Code Quality Problems - What Worked and What not?. In *2021 International Conference on Code Quality (ICCQ)*. 1–5. https://doi.org/10.1109/ICCQ51190.2021.9392977

Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. 2020. An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations. *IEEE Access* 8 (2020), 72239–72260. https://doi.org/10.1109/ACCESS.2020.2987631

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 404–415. https://doi.org/10.1109/ICSE.2017.44

Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. *CoRR* abs/1803.03806 (2018). arXiv:1803.03806 http://arxiv.org/abs/1803.03806

Evgeniy Ryzhkov. 2011. How to add a new diagnostic rule into PVS-Studio? Days from developers' life. https://pvs-studio.com/en/blog/posts/0110/

Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., USA.

Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 515–527. https://doi.org/10.1145/3236024.3236034

Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 6117–6124. https://doi.org/10.24963/ijcai.2019/847

Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active Inductive Logic Programming for Code Search. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 292–303. https://doi.org/10.1109/ICSE.2019.00044

Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 130–140. https://doi.org/10.1109/SANER.2018.8330203

Armando Solar Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html

Amann Sven and Hoan Anh Nguyen. 2019. MUDetect : An API-Misuse Detector. https://github.com/stg-tud/MUDetect.

Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. *Example-Guided Synthesis of Relational Queries*. Association for Computing Machinery, New York, NY, USA, 1110–1125. https://doi.org/10.1145/3453483.3454098

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 832–837. https://doi.org/10.1145/3238147.3240732

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ByloJ20qtm

Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic Programming by Example with Pre-Trained Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 100 (oct 2021), 25 pages. https://doi.org/10.1145/3485477

Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. 2021. Data-Driven Synthesis of Provably Sound Side Channel Analyses. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 810–822. https://doi.org/10.1109/ICSE43902.2021.00079

Junchi Yan, Yu Tian, Hongyuan Zha, Xiaokang Yang, Ya Zhang, and Stephen M. Chu. 2013. Joint Optimization for Consistent Multiple Graph Matching. In *2013 IEEE International Conference on Computer Vision*. 1649–1656. https://doi.org/10.1109/ICCV.2013.207

Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISAN: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 363–378.