Verifying Data Constraint Equivalence in FinTech Systems

Chengpeng Wang*, Gang Fan†, Peisen Yao‡, Fuxiong Pan†, and Charles Zhang*
*Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China.
Email: {cwangch, charlesz}@cse.ust.hk,
†Ant Group, Shenzhen, China. Email: {fangang, fuxiong.pfx}@antgroup.com,
‡Zhejiang University, Hangzhou, China. Email: pyaoaa@zju.edu.cn

***Abstract*—Data constraints are widely used in FinTech systems for monitoring data consistency and diagnosing anomalous data manipulations. However, many equivalent data constraints are created redundantly during the development cycle, slowing down the FinTech systems and causing unnecessary alerts. We present EQDAC, an efficient decision procedure to determine the data constraint equivalence. We first propose the symbolic representation for semantic encoding and then introduce two light-weighted analyses to refute and prove the equivalence, respectively, which are proved to achieve in polynomial time. We evaluate EQDAC upon 30,801 data constraints in a FinTech system. It is shown that EQDAC detects 11,538 equivalent data constraints in three hours. It also supports efficient equivalence searching with an average time cost of 1.22 seconds, enabling the system to check new data constraints upon submission.**

***Index Terms*—Equivalence Verification, Data Constraints, FinTech Systems**

I. Introduction

With the rapid development of E-commerce, FinTech systems have become increasingly essential to industrial production. They consist of a cluster of database-backed applications manipulating large amounts of sensitive data [1]. Any incorrect data value can yield system misbehaviors and cause immeasurable financial losses. To ensure reliability, it is a common practice to specify target properties as data constraints [2], [3] for runtime checking. If a data constraint is violated, developers can receive an alert for further diagnosis.

Unfortunately, the continuous submissions from developers make data constraints accumulate rapidly and can even introduce redundancy. In a global FinTech company A, 103 developers submitted 2,306 data constraints in the first quarter of 2022. Unaware of previous submissions, they create equivalent data constraints, which gradually become the technical debt [4], wasting computing resources and increasing the burden of system maintenance. To resolve the redundancy, the developers expect to search the existing equivalent data constraints before submitting new ones, thereby avoiding redundant submissions. Besides, quality assurance teams are eager to examine data constraint repositories regularly, seeking more opportunities for optimization based on the equivalence relation. Thus, it is relevant to verify the data constraint equivalence for better maintenance in a FinTech system.

Goal and Challenges. We aim to design a decision procedure determining whether two data constraints are equivalent. However, it is stunningly challenging to find a solution fitting industrial requirements. First, the decision procedure should be highly efficient, as FinTech systems often contain tens of thousands of data constraints, which amplify the efficiency bottleneck greatly. Any inefficiency in the decision procedure can result in significant burdens of adoption. Second, it is crucial to guarantee soundness and prove the equivalence as completely as possible. Otherwise, it would remove necessary data constraints or miss equivalent ones, resulting in financial losses or hiding opportunities for further optimization, respectively. In reality, data constraints contain various operations upon different data types, increasing the difficulty of achieving these objectives simultaneously.

Existing Efforts. There have been two lines of research on equivalence verification. One line of the techniques leverages the specified rewrite rules and checks whether a program can be transformed to the other via *term rewriting* [5], [6]. Although the rewrite rules theoretically ensure soundness, they can only identify restrictive forms of equivalent patterns [7], and the vast search space of applying rewrite rules also brings great overhead [8]. The other line encodes the program semantics with logical formulas and performs the symbolic reasoning by invoking an SMT solver [9]–[11]. It provides a general approach to verify the equivalence, while an SMT solver is not efficient enough to reason a large number of data constraints. The solver has to be invoked thousands of times in the equivalence clustering and searching, accumulating the overhead and finally degrading the overall efficiency [12].

Insight and Solution. Our key idea originates from two critical observations. First, non-equivalent data constraints often contain different variables, literals, or operators. For example, the data constraint in Fig. 1a examines the attributes *oid* and *in* in the table *t*, while the data constraint in Fig. 1c examines the attributes *iid* and *new* instead. The lexical differences guide the generation of concrete values to make two data constraints evaluate differently. Second, equivalent data constraints often converge towards similar syntactic structures. For instance, the data constraints in Fig. 1b and Fig. 1d only differ in the orders of assertions, branches, and commutative operands after

```
s = 'IN';
if(contains(t.ty,s))
  assert(t.in > 0);
else
  assert(t.out > 0);
assert(t.amt > 0);
assert(t.oid != 0);
```

(a)

```
if(contains(t.ty,'IN')){
  assert(t.old == t.new - t.in);
} else {
  assert(t.old == t.new + t.out);
}
assert(t.oid != 0);
assert(t.iid != 0);
```

(b)

```
s = 'IN';
if(not contains(t.ty,s))
  assert(t.out > 0);
else
  assert(t.new > 0);
assert(t.amt > 0);
assert(t.iid != 0);
```

(c)

```
assert(t.iid != 0);
assert(t.oid != 0);
if(not contains(t.ty,'IN'))
  cash = t.out + t.new;
else
  cash = t.new - t.in;
assert(cash == t.old);
```

(d)

Fig. 1: Examples of data constraints

eliminating user-defined variables. The isomorphic syntactic structures are the witness of their equivalence. Thus, we can leverage the lexical differences and syntactic isomorphism to efficiently refute and prove the equivalence, respectively, avoiding unnecessary SMT solving for better performance.

Based on the insight, we present EQDAC, an efficient decision procedure for the equivalence verification. We establish a first-order logic (FOL) formula as the symbolic representation to depict the semantics. To refute the equivalence, we perform the *divergence analysis* to explore the symbolic representations and generate the concrete values of variables, which simultaneously make one data constraint hold and the other violated. To prove the equivalence, we conduct the *isomorphism analysis* with a tree isomorphism algorithm [13] to examine whether the two symbolic representations can be transformed into each other by reordering the clauses and commutative terms. We combine the two analyses with the SMT solving, which determines the logical equivalence of the symbolic representations, finally obtaining a three-staged decision procedure.

We implement EQDAC and evaluate it upon a FinTech system in Company A, which maintains 30,801 data constraints in total. Leveraging EQDAC, we discover that 11,538 data constraints have at least one equivalent variant in the system, indicating that 7,842 data constraints are redundant. EQDAC finishes the equivalence clustering in three hours and achieves the equivalence searching in 1.22 seconds per data constraint. Except for the SMT solving, the stages of EQDAC can be proven to work in polynomial time. We also prove the soundness and completeness of EQDAC theoretically for a given syntax of data constraints. Our efforts have crossed the line from developing an academic-only decision procedure to one that is practical enough to be deployed.

In summary, we make the following major contributions:

- We formulate the data constraint equivalence problem, which is critical for the FinTech system maintenance.
- We propose a sound and complete decision procedure EQDAC to determine the data constraint equivalence, efficiently supporting the equivalence clustering and searching.

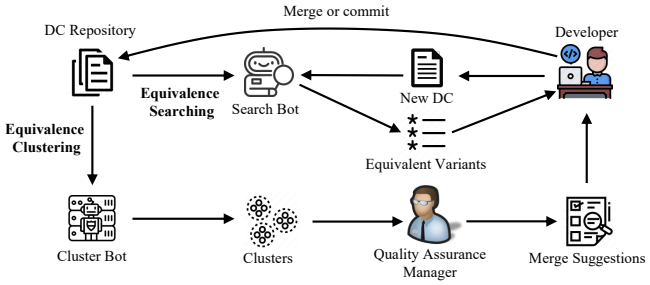


Fig. 2: The workflow of equivalence searching and clustering

- We implement EQDAC and evaluate it upon the data constraints in a global FinTech company with 1 billion active users, showing that it efficiently detects a significant number of equivalent data constraints.

II. Background and Motivation

This section presents the background and highlights the motivation of our work.

A. Equivalent Data Constraints in FinTech Systems

A FinTech system usually consists of a cluster of database-backed applications manipulating large amounts of data, making the data records exhibit specific properties. To improve the reliability, the developers often specify data constraints to describe target properties and set up a data reconciliation (DR) platform [14] to examine them in the runtime. Once a data constraint is violated, developers can receive detailed runtime information to guide further system diagnosis.

In reality, many development teams continuously submit data constraints to a central DR platform. For example, around 100 teams in Company A actively submit data constraints daily to the platform. Unaware of existing submissions, developers often submit data constraints equivalent to existing ones. Besides, the developers tend to be conservative about removing constraints, as they do not want to risk missing data errors. Finally, the accumulation of equivalent data constraints becomes the technical debt [4] of a FinTech system:

- The DR platform examines equivalent data constraints redundantly, which causes unnecessary resource consumption, e.g., CPU time, disk IO, and network traffic.
- Multiple alerts are fired to developers if equivalent data constraints are violated, which requires more time budget for diagnosis, lengthening the system's development cycle.
- All the equivalent data constraints should be updated if the table schema or the target properties are changed, involving extra labor in interacting with the DR platform.

Thus, it is crucial to tackle equivalent data constraints in the maintenance, which promotes resource-saving and eases the debugging and refactoring of a FinTech system.

B. Resolving Equivalent Data Constraints

To resolve the technical debt, the developers of Company A propose two demands, namely equivalence clustering and

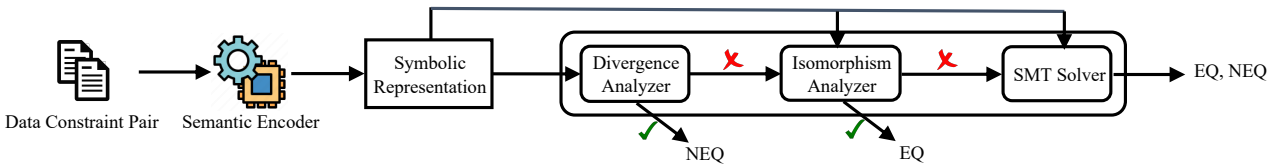


Fig. 3: Schematic overview of our decision procedure EQDAC

equivalence searching, to tackle equivalent data constraints. Specifically, they expect to integrate two bots into the CI/CD workflow [15] of a FinTech system as follows.

- *Equivalence searching*: A developer commits a new data constraint to the bot for searching existing equivalent variants. The list of equivalent variants assists the developer in deciding whether to merge it with any existing one. The workflow is shown in the upper part of Fig. 2.
- *Equivalence clustering*: A quality assurance (QA) manager exports all the data constraints to the bot, which divides the data constraints into equivalence clusters. Then, the QA manager summarizes merge suggestions and sends them to developers for further confirmation. The lower part of Fig. 2 shows the workflow of equivalence clustering.

Generally, the two bots resolve the redundancy from two perspectives, respectively. First, the equivalence searching conducts the instant checking of newly-submitted data constraints, enabling the developers to avoid redundancy if possible. Second, the equivalence clustering supports the nightly scan of the whole repository of data constraints. The QA managers can inspect the clustering information to find opportunities for merging equivalent ones. During the development cycle, the two bots can serve as two lines of defense for redundancy issues in the CI/CD workflow.

To automate the overall workflow, we need an efficient decision procedure to verify whether two data constraints are equivalent or not. Specifically, the two bots would invoke the decision procedure to determine the equivalence in the clustering and searching, respectively. In this work, we aim to design an effective solution for verifying data constraint equivalence, and promoting data constraint maintenance with our decision procedure.

III. EQDAC IN A NUTSHELL

This section presents a motivating example to show our insight (§ III-A) and outlines our decision procedure (§ III-B).

A. Motivating Examples

Verifying the data constraint equivalence is non-trivial in industrial scenarios. First, the cost of the decision procedure can accumulate significantly due to the vast number of data constraints [16]. Second, the decision procedure can prune necessary data constraints or miss equivalent ones if it is not sound or complete, increasing the risk of data security and hiding the opportunity for optimization. Thus, we need to simultaneously ensure the soundness, completeness, and efficiency of the decision procedure.

Fig. 1 shows four data constraints as examples. Specifically, the data constraints in Fig. 1a and Fig. 1c depict the properties where three attributes of the table t have positive values in two cases, and the values of *oid* and *iid* are not 0, respectively. Besides, the data constraints in Fig. 1b and Fig. 1d describe the property where the changes to the account balances are equal to the transferred cash amount, and the ids of the two accounts, i.e., iid and oid, are not 0. According to the examples, we can obtain the following two important observations:

- Non-equivalent data constraints tend to have different lexical tokens, such as database attributes, literals, and operators. For example, the data constraints in Fig. 1a and Fig. 1c examine different attributes. It is likely to generate the values of table attributes, making them evaluate differently.
- Equivalent data constraints often only differ in the orders of commutative operands and independent statements after eliminating user-defined variables. For instance, the data constraints in Fig. 1b and Fig. 1d share the isomorphic syntactic structure, which implies their equivalence.

Based on the observations, we realize that the lexical differences and syntactic isomorphism enable us to efficiently refute and prove the equivalence, respectively. If we generate "good" concrete values making two data constraints evaluate differently or find the isomorphism between the syntactic structures, we can avoid SMT solving and achieve high efficiency.

B. Outline of Decision Procedure

According to our insight, we design an efficient, sound, and complete decision procedure for verifying data constraint equivalence. To depict the data constraint semantics, we propose the semantic encoding to construct a FOL formula in a restrictive form as its symbolic representation, which eliminates user-defined variables (e.g., the variable *cash* in Fig. 1d). Based on the symbolic representations, our decision procedure works in three stages, as shown in Fig. 3.

- The divergence analysis explores the symbolic representations with the guidance of lexical differences, aiming to generate concrete values that make data constraints evaluate differently. For example, it explores the clause induced by the last assertion in Fig. 1a and assigns 0 to the attribute *oid* to violate the assertion. Also, it concretizes the variables in Fig. 1c to make the data constraint satisfied.
- The isomorphism analysis constructs the parse trees of the symbolic representations and examines whether the parse trees are isomorphic. The analysis abstracts away the order of commutative constructs, such as independent statements and commutative operands. For example, it discovers the

$$
\begin{aligned}
&\mathcal{V} := v_d \mid x \\
&\mathcal{L} := \{l_i \mid i \geq 1\} \\
&\mathcal{A} := l \mid v_d \mid a_1 \oplus a_2 \\
&\mathcal{C} := a_1 \odot a_2 \mid x_1 \odot x_2 \mid a \odot x \mid x \odot a \mid p(v, l) \mid p(v_1, v_2) \\
&\mathcal{B} := c \mid b_1 \textbf{ and } b_2 \mid b_1 \textbf{ or } b_2 \mid \textbf{not } b \mid \textbf{ite}_b(c_0, b_1, b_2) \\
&\mathcal{S} := x = a \mid \textbf{assert}(b) \mid s_1; s_2 \mid \textbf{ite}_s(c_0, s_1, s_2) \\
&\mathcal{R} := s+ \\
&\oplus := \ + \mid - \mid \times \mid \div \\
&\odot := \ > \mid < \mid \geq \mid \leq \mid == \mid \neq \\
&\mathcal{P} := \{\text{prefixOf},\ \text{suffixOf},\ \text{contains},\ \text{equals}\}
\end{aligned}
$$

Fig. 4: The syntax of data constraints

isomorphic structures in Fig. 1b and Fig. 1d, blurring the orders of assertions and the operands of $+$ and $==$.

- If the first two analyses can not refute or prove the equivalence, we invoke an SMT solver to check the logical equivalence of the symbolic representations. To ensure soundness and completeness, we perform the SMT encoding with a decidable fragment in the combined theory of bit-vector, floating-point arithmetic, and string.

Apart from soundness and completeness, EQDAC also features a theoretical guarantee of complexity. The symbolic representation construction, the divergence analysis, and the isomorphism analysis can work in polynomial time to the size of the abstract syntax tree of a data constraint. Our evaluation also provides strong evidence of the EQDAC's high efficiency in the equivalence clustering and searching.

IV. Problem Formulation

This section presents the syntax (§ IV-A) and formulates the data constraint equivalence problem (§ IV-B).

A. Data Constraint Syntax

Fig. 4 summarizes the syntax. A *variable* is a data variable $v_d \in \mathcal{V}_d$ indicating the value of a table attribute, or an user-defined variable $x \in \mathcal{V}_u$ storing the value temporally. Its value can be a finite-length integer, a floating point number, or a string. A *literal* is a constant value. An *arithmetic expression* can be a literal, a data variable, or a compound arithmetic expression. A *comparison expression* compares arithmetic expressions and user-defined variables, or examines the strings with predicates $p \in \mathcal{P}$. A *Boolean expression* is a comparison expression or a compound expression with logical connectives. A *statement* is an assignment, an assertion, a sequencing, or an $\textbf{ite}_s$ statement. Particularly, the conditions in $\textbf{ite}_b$ expressions and $\textbf{ite}_s$ statements only relate to data variables. Finally, a *data constraint* consists of finite statements. All its assertions are expected to hold for given database tables.

The syntax is expressive enough to specify target properties in real-world scenarios. It covers all the patterns in [3], such as value comparison, conditional comparison, etc. Also, user-defined variables support writing data constraints flexibly. Arithmetic operations and string predicates support expressing complex properties, e.g., comparing the sums of cash amounts and matching between string variables.

B. Data Constraint Equivalence Problem

Before stating the problem, we first introduce the notions of interpretation and semantic equivalence as follows.

Definition 1. An interpretation I maps each data variable v_d to a value in its domain. I is a model of a data constraint r, denoted by $I \models r$, if all the assertions hold under I.

Example 1. The following interpretation I is a model of the data constraint in Fig. 1a.

$$I = [t.ty \mapsto \text{'IN'}, t.in \mapsto 1, t.out \mapsto 0, t.oid \mapsto 1, t.amt \mapsto 1]$$

An interpretation indicates the values of table attributes. A data constraint induces a set of interpretations making its assertions hold. Formally, we define the semantic equivalence.

Definition 2. The data constraints r_1 and r_2 are semantically equivalent, denoted by $r_1 \simeq r_2$, if and only if

$$\forall I : I \models r_1 \Leftrightarrow I \models r_2$$

Example 2. Based on Example 1, we can construct

$$I' = I[t.new \mapsto 0, t.iid \mapsto 1]$$

I' is not a model of the data constraint in Fig. 1c, while it is a model of the data constraint in Fig. 1a, indicating that they are not semantically equivalent.

In this work, we aim to propose a decision procedure to verify whether r_1 is semantically equivalent to r_2 for a given data constraint pair (r_1, r_2). However, finding a sound, complete, and efficient solution is challenging. Theoretically, any instance of SAT problem [17] can be reduced to an instance of our problem by constructing two proper data constraints in polynomial time. Formally, we state the complexity barrier of our problem as follows.

Theorem 1. Data constraint equivalence problem is NP-hard.

Roadmap. To verify the equivalence, we propose a symbolic representation to encode the semantics (§ V) and design an efficient decision procedure (§ VI). Particularly, we introduce light-weighted reasoning to refute and prove the equivalence efficiently, which is our main technical contribution. By fusing our light-weighted reasoning with SMT-based analysis, our decision procedure features soundness and completeness, and achieves high efficiency in supporting the equivalence clustering and searching.

V. Semantic Encoding

This section introduces the symbolic representation to depict the semantics (§ V-A), presents the symbolic evaluation (§ V-B), and summarizes the benefit at the end (§ V-C).

A. Symbolic Representation

A data constraint is essentially a program with data variables as inputs. The values of data variables determine the values of all the variables and expressions. Based on the intuition, we propose the concepts of symbolic terms and conditions to depict the values of variables and expressions.

Definition 3. A symbolic term τ represents the value of a variable or a literal in either of the forms:

- $\tau := v_d$ or $\tau := l$ is a data variable or a literal, respectively.
- $\tau := \tau_1 \oplus \tau_2$ is a compound term with an arithmetic operator.

Definition 4. A symbolic condition ϕ is a FOL formula in one of the following forms:

- An atomic condition is an arithmetic comparison of two symbolic terms or a string comparison, i.e., $\phi := \tau_1 \odot \tau_2$ or $\phi := p(\tau_1, \tau_2)$, where $p \in \mathcal{P}$ is a string predicate.
- A compound condition is a FOL formula with logical connectives, i.e., $\phi := \phi_1 \wedge \phi_2$, $\phi := \phi_1 \vee \phi_2$, or $\phi := \neg\phi_0$.

Example 3. In Fig. 1d, the values of *cash* can be represented by the terms $t.out + t.new$ and $t.new - t.in$. The condition of the $\textbf{ite}_s$ statement is encoded by $\neg\texttt{contains}(t.ty,$ 'IN').

The symbolic terms represent the values of variables, literals, and arithmetic expressions, while the symbolic conditions encode the values of Boolean expressions, providing the ingredient for defining the symbolic representations.

Definition 5. For a data constraint r, its symbolic representation is a symbolic condition φ satisfying

- For any interpretation I, $I \models r$ if and only if $I \models \varphi$.
- The negations only occur before string atomic constraints.

Intuitively, the symbolic representation encodes the semantics faithfully with a FOL formula, which only relates to data variables and exclude redundant negations. It abstracts away user-defined variables and blurs syntactic differences in terms of negations effectively, enabling us to design light-weighed reasoning for equivalence verification. In what follows, we show how to construct the symbolic representation in detail.

B. Symbolic Evaluation

Now we propose the symbolic evaluation to construct the symbolic representation. Basically, the symbolic evaluation consists of two stages, which collects the values of Boolean expressions in each assertion, and eliminates unnecessary negations, respectively. Before delving into details, we first introduce the notion of the symbolic state.

Definition 6. Given a data constraint r, the symbolic state $\mathbf{S}$ at program location ℓ is $(\mathbf{E}, \mathbf{\Phi})$, where

- An *environment* $\mathbf{E}$ maps a variable v or an arithmetic expression e to a term-condition pair set $\{(\tau, \phi)\}$, indicating that v or e evaluates to the same value of τ when ϕ holds.
- A *property* $\mathbf{\Phi}$ is a symbolic condition that summarizes the assertions in r before the program location ℓ.

Example 4. After the first assertion in Fig. 1d, we have

$$\mathbf{E} = [t.iid \mapsto \{(t.iid, T)\},\ 0 \mapsto \{(0, T)\}] \quad \mathbf{\Phi} = (t.iid \neq 0)$$

Now we present the technical details of the symbolic evaluation. In the first stage, we evaluate the variables and expressions to obtain a FOL formula depicting the semantics, which only relates to the data variables. Specifically, we define the evaluation rules in Fig. 5 and Fig. 6.

$$\texttt{ASSIGN}\ \frac{\mathbf{E} \vdash_e a \rightsquigarrow V \quad \mathbf{E}' = \mathbf{E}[v \mapsto V]}{\mathbf{E}, \mathbf{\Phi} \vdash v = a \rightsquigarrow \mathbf{E}', \mathbf{\Phi}}$$

$$\texttt{ASSERT}\ \frac{\mathbf{E} \vdash_b b \rightsquigarrow \psi \quad \mathbf{\Phi}' = \mathbf{\Phi} \wedge \psi}{\mathbf{E}, \mathbf{\Phi} \vdash \textbf{assert}(b) \rightsquigarrow \mathbf{E}, \mathbf{\Phi}'}$$

$$\texttt{SEQ}\ \frac{\mathbf{S} \vdash s_1 \rightsquigarrow \mathbf{S}_1 \quad \mathbf{S}_1 \vdash s_2 \rightsquigarrow \mathbf{S}'}{\mathbf{S} \vdash s_1; s_2 \rightsquigarrow \mathbf{S}'}$$

$$\texttt{ITE-S}\ \frac{\begin{array}{c}\mathbf{E} \vdash_b c_0 \rightsquigarrow \gamma_1 \quad \gamma_2 = \neg\gamma_1 \quad \mathbf{E}, \mathbf{\Phi} \vdash s_i \rightsquigarrow \mathbf{E}_i, \mathbf{\Phi}_i \\ \mathbf{E}' = [u \mapsto \bigcup_{i=1}^{2} \{(\tau_i, \phi_i \wedge \gamma_i) | (\tau_i, \phi_i) \in \mathbf{E}_i(u)\}]\end{array}}{\mathbf{E}, \mathbf{\Phi} \vdash \textbf{ite}_s(c_0, s_1, s_2) \rightsquigarrow \mathbf{E}', ite(\gamma_1, \mathbf{\Phi}_1, \mathbf{\Phi}_2)}$$

Fig. 5: Evaluation rules of statements

$$\texttt{VAR}\ \frac{u \in \mathcal{L} \cup \mathcal{V}_d \quad U = \{(u, T)\}}{\mathbf{E} \vdash_e u \rightsquigarrow U}$$

$$\texttt{AE}\ \frac{\begin{array}{c}a_i \in \mathcal{A} \quad \mathbf{E} \vdash_e a_i \rightsquigarrow U_i \\ A = \{(t_1 \oplus t_2, \phi_1 \wedge \phi_2) \mid (t_i, \phi_i) \in U_i\}\end{array}}{\mathbf{E} \vdash_e a_1 \oplus a_2 \rightsquigarrow A}$$

$$\texttt{ACmp}\ \frac{\begin{array}{c}u_i \in \mathcal{A} \cup \mathcal{V}_u \quad \mathbf{E} \vdash_e u_i \rightsquigarrow U_i \\ B = \{(t_1 \odot t_2) \wedge \phi_1 \wedge \phi_2 \mid (t_i, \phi_i) \in U_i\}\end{array}}{\mathbf{E} \vdash_b u_1 \odot u_2 \rightsquigarrow \bigvee_{\phi \in B} \phi}$$

$$\texttt{ITE-E}\ \frac{\mathbf{E} \vdash_b c_0 \rightsquigarrow \gamma_0 \quad \mathbf{E} \vdash_b b_i \rightsquigarrow \gamma_i}{\mathbf{E} \vdash_b \textbf{ite}_b(c_0, b_1, b_2) \rightsquigarrow (\gamma_1 \wedge \gamma_0) \vee (\gamma_2 \wedge \neg\gamma_0)}$$

Fig. 6: Helper rules evaluating expressions

- The rule `ASSIGN` evaluates the RHS with the rules `VAR` and `AE` in Fig. 6, and applies the strong update to $\mathbf{E}$, enforcing the user-defined variable v and the expression a have the same value. It successfully evaluates user-defined variables, making the symbolic terms only relate to the data variables.
- The rule `ASSERT` evaluates the Boolean expression b to a symbolic condition ψ. It then connects ψ and the original property $\mathbf{\Phi}$ with a logical conjunction. This, in turn, forms a property that accumulates the conditions of the assertions.
- The rules `SEQ` and `ITE-S` are defined straightforwardly. `SEQ` applies the evaluation rules of two components sequentially. `ITE-S` evaluates the two cases separately and joins two symbolic states according to the branch condition.

We omit the rules of evaluating string comparisons and other compound Boolean expressions due to limited space, which are similar to the rules `ACmp` and `ITE-E`. Based on the rules, we evaluate a data constraint stepwise. Initially, the symbolic state is a pair of empty mapping and a *true* value. By applying the rule of each statement along control flow paths, we obtain the symbolic state at each program location and finally summarize all the assertions with the property $\mathbf{\Phi}_e$ at the exit, which depicts the semantics of the data constraint.

Example 5. Consider the data constraint in Fig. 1d. We obtain $\mathbf{\Phi} = \phi_1 \wedge ((\phi_2 \wedge \phi_4) \vee (\phi_3 \wedge \neg\phi_4))$ at its exit, where

$$\phi_1 = (t.iid \neq 0) \wedge (t.oid \neq 0)\ \ \phi_2 = (t.out + t.new = t.old)$$
$$\phi_3 = (t.new - t.in = t.old)\ \ \phi_4 = \neg\texttt{contains}(t.ty,\ \text{'IN'})$$

In the second stage, we eliminate the negations in $\mathbf{\Phi}_e$ that do not apply to atomic string constraints. Technically, we first transform $\mathbf{\Phi}_e$ into the negation normal form (NNF), in which the negation applies only to atomic formulas. Then, we eliminate the negation before each atomic arithmetic constraint by changing the comparison operator, e.g., transforming $\neg(t.a \geq t.b)$ to $t.a < t.b$. Notably, the above transformations can be achieved by the breadth-first search upon the parse tree of $\mathbf{\Phi}_e$, where the symbolic representation is constructed on the fly. The overall time complexity is linear to the size of $\mathbf{\Phi}_e$.

Example 6. In Example 5, we eliminate the negations and get the symbolic representation $\varphi = \phi_1 \wedge ((\phi_2 \wedge \phi_4) \vee \phi')$, where $\phi' = (t.new - t.in = t.old) \wedge \texttt{contains}(t.ty,\ \text{'IN'})$.

C. Summary

The symbolic representation is essentially a Boolean function of data variables, featuring the following three benefits:

- The symbolic representations preserve the lexical differences in terms of data variables, literals, and operators, which can indicate the possible non-equivalence.
- The symbolic evaluation evaluates the user-defined variables, abstracting away the difference in terms of their names, which do not affect the semantics.
- The elimination of unnecessary negations normalizes the FOL formulas and yields isomorphic symbolic representations for more equivalent data constraints.

Thus, the semantic encoding exposes lexical differences and syntactic isomorphism for light-weight reasoning, which efficiently refutes and proves the equivalence (§ VI-A § VI-B).

VI. Decision Procedure

In this section, we first introduce the divergence analysis (§ VI-A) and isomorphism analysis (§ VI-B) for efficiently refuting and proving the equivalence, respectively. We then combine the two analyses with SMT solving to establish the decision procedure (§ VI-C). In what follows, we denote the data constraints by r_1 and r_2 and their symbolic representations by φ_1 and φ_2 for demonstration.

A. Divergence Analysis

Based on Definition 5, φ_1 and φ_2 depict the semantics of two data constraints faithfully. We can safely refute the equivalence if there exists an interpretation I making them evaluate to different truth values. However, it is non-trivial to obtain such a desired interpretation efficiently. The random sampling may hit a desired interpretation successfully after failing many attempts, which can degrade the efficiency significantly. To resolve the problem, we attempt to explore specific Boolean structures of φ_1 and φ_2 and concretize the data variables within the structures. Formally, we introduce the *degrees of freedom* to guide the exploration.

Definition 7. For two symbolic representations φ_1 and φ_2, the degrees of freedom of a clause ϕ occurring in φ_1 is

$$\mathcal{DF}(\phi \mid \varphi_1, \varphi_2) = \frac{1}{h(\phi)} \cdot \sum_{M \in \{V_d, L, O\}} |M(\phi) \setminus M(\varphi_2)|$$

Algorithm 1: Divergence analysis

Input: φ_1, φ_2: Two symbolic representations;
Output: Whether $\exists I : \neg(I \models \varphi_1 \leftrightarrow I \models \varphi_2)$

```
1  foreach (φ1, φ2) ∈ {(φ1, φ2), (φ2, φ1)} do
2      I ← ⊥;
3      status ← T;
4      explore(φ1, φ1, φ2, F);
5      explore(φ2, φ2, φ1, T);
6      if status is T then
7          return true;
8  return unknown;
9  Procedure explore(φ, φ, φ', tv)
10     if status then
11         if φ is atomic then
12             if FreeVar(φ, I) ≠ ∅ then
13                 I ← concretize(φ, tv);
14             else
15                 status ← check(I ⊨ φ = tv);
16         else if (LC(φ), tv) ∈ {(∧, T), (∨, F)} then
17             foreach φ_i ∈ C(φ) do
18                 explore(φ_i, φ, φ', tv);
19         else
20             φ' ← arg max_{φ_i ∈ C(φ)} DF(φ_i | φ, φ');
21             explore(φ', φ, φ', tv);
```

$h(\psi)$ is the height of the parse tree of ψ. $V_d(\psi)$ contains the data variables in ψ but excludes arithmetic operands. $L(\psi)$ and $O(\psi)$ contain the literals and operators in ψ, respectively.

Intuitively, a larger degrees of freedom indicates a higher possibility of making ϕ evaluate to a target truth value:

- First, a smaller value of $h(\phi)$ indicates the opportunity of finding the desired interpretation with fewer explorations.
- Second, a larger value of $|M(\phi) \setminus M(\varphi_2)|$ indicates that ϕ has more unique lexical tokens absent in φ_2. The concretizations of data variables in ϕ and φ_2 are less intertwined.
- Third, $V_d(\phi)$ excludes arithmetic operands, as arithmetic operations can increase the difficulty of concretization.

Example 7. Consider the data constraints in Fig. 1a and Fig. 1c. According to § V, their symbolic representations are

$$\varphi_1 = ((t.in > 0 \wedge \phi_c) \vee (t.out > 0 \wedge \neg\phi_c)) \wedge \phi_a \wedge \phi_o$$
$$\varphi_2 = ((t.out > 0 \wedge \neg\phi_c) \vee (t.new > 0 \wedge \phi_c)) \wedge \phi_a \wedge \phi_i$$

where $\phi_a = (t.amt > 0)$, $\phi_o = (t.oid \neq 0)$, $\phi_i = (t.iid \neq 0)$, and $\phi_c = \texttt{contains}(t.ty,\ \text{'IN'})$. Let ϕ denote the first clause of φ_1. We have $V_d(\phi) \setminus V_d(\varphi_2) = \{t.in\}$, $L(\phi) \subseteq L(\varphi_2)$, and $O(\phi) \subseteq O(\varphi_2)$ Thus, we have $\mathcal{DF}(\phi \mid \varphi_1, \varphi_2) = \frac{1}{3}$. Similarly, we have $\mathcal{DF}(\phi_a \mid \varphi_1, \varphi_2) = 0$ and $\mathcal{DF}(\phi_o \mid \varphi_1, \varphi_2) = 1$.

Based on the degrees of freedom, we propose the divergence analysis to generate a desired interpretation. Alg. 1 shows its technical details. It receives two symbolic representations φ_1 and φ_2 and attempts to generate a desired interpretation enforcing them evaluate differently (lines 1–7). The function

explore traverses the clauses level by level (lines 9–21), handling three kinds of clauses ϕ with specific strategies:

- If ϕ is atomic, we concretize the free variables to make ϕ evaluate to tv (lines 12–13). If there is no free variable, we check whether ϕ evaluates to tv under I (line 15).
- If ϕ is a connected with $\wedge$ and tv is *true*, or ϕ is connected with $\vee$ and tv is *false*, we explore all the clauses in ϕ and enforce them evaluates to tv (lines 16–18).
- Otherwise, we select the clause ϕ' with the maximal degrees of freedom and enforce it evaluate to tv (lines 20–21).

If each clause evaluates to the target value, Alg. 1 finds the desired interpretation, thereby refuting the equivalence.

Example 8. In Example 7, φ_1 is connected with the logical conjunction. We only need to select and explore one of its clauses if we want to make φ_1 evaluate to *false*. The third clause ϕ_o has a larger degrees of freedom than the other two, so we select it and assign 0 to $t.oid$. Similarly, we can enforce φ_2 evaluate to *true*, which finally refutes the equivalence.

Lastly, it is worth mentioning that the data constraints with different lexical tokens are often non-equivalent, while it is unsound to refute the equivalence directly based on lexical differences. In contrast, our divergence analysis essentially utilizes lexical differences to guide the interpretation generation, which supports refuting the equivalence soundly.

B. Isomorphism Analysis

As the FOL formulas, the symbolic representations φ_1 and φ_2 are logically equivalent if we can transform φ_1 to φ_2 by reordering commutative sub-formulas and terms in φ_1. For example, the evaluation of a FOL formula does not depend on the order of the clauses connected with the logical disjunction and conjunction. Also, any permutation of the operands of commutative arithmetic operators, such as addition and multiplication, always yields the logically equivalent formula. In other words, we can prove the data constraint equivalence safely by identifying the isomorphism between φ_1 and φ_2.

Based on the above key idea, we propose the isomorphism analysis to determine whether the parse trees of φ_1 and φ_2 are isomorphic, which is formulated in Alg. 2. Using the AHU algorithm [13] for tree isomorphism checking, Alg. 2 proves the data constraint equivalence if the parse trees are isomorphic (lines 1–2). Particularly, the functions *SCT* and *STT* process the clauses and terms of a symbolic representation in a top-down manner, respectively, creating tree nodes and leaf nodes in the parse tree.

- When processing a non-atomic formula φ, *SCT* creates a tree node to store the logical connective, and appends all the parse trees of its clauses (lines 5–6).
- For an atomic condition, *SCT* creates a tree node if the comparison operator is in $\{=, \neq\}$ or the string predicate is *equals* (lines 8–9). Otherwise, it adds a leaf node to make sub-trees nonexchangeable (lines 10–14). Notably, it normalizes inequalities to enforce them using $>$ and $\geq$ only, which supports discovering more equivalent inequalities.

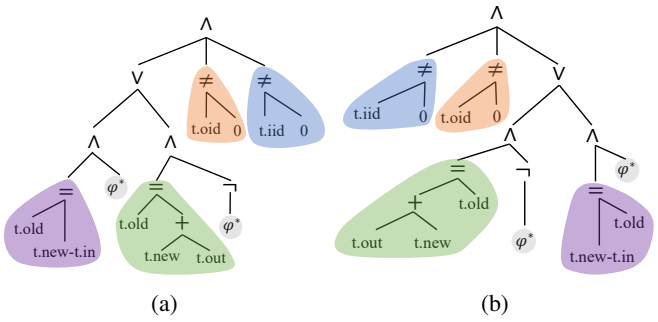
Algorithm 2: Isomorphism analysis

Input: φ_1, φ_2: Two symbolic representations;
Output: Whether $\forall I : I \models \varphi_1 \leftrightarrow I \models \varphi_2$

1 **if** `AHUcheck` $(SCT(\varphi_1), SCT(\varphi_2))$ **then**
2 **return** *true*;
3 **return** *unknown*;
4 **Procedure** `SCT` (ϕ)
5 **if** $\phi : \phi_1 \circledast \cdots \circledast \phi_k$ ***and*** $\circledast \in \{\wedge, \vee, \neg\}$ **then**
6 **return** $\texttt{Tree}(\circledast, \{SCT(\phi_i) \mid 1 \leq i \leq k\})$;
7 **else if** $\phi : \tau_1 \circledast \tau_2$ ***or*** $\phi : \circledast(\tau_1, \tau_2)$ **then**
8 **if** $\circledast \in \{=, \neq, equals\}$ **then**
9 **return** $\texttt{Tree}(\circledast, \{STT(\tau_1), STT(\tau_2)\})$;
10 **else if** $\circledast \in \{<, \leq\}$ **then**
11 $\circledast' \leftarrow \texttt{flip}(\circledast)$;
12 **return** $\texttt{Leaf}(\circledast', STT(\tau_1), STT(\tau_2))$;
13 **else**
14 **return** $\texttt{Leaf}(\circledast, STT(\tau_1), STT(\tau_2))$;
15 **Procedure** `STT` (τ)
16 **if** $\texttt{Op}(\tau) = \{\circledast\}$ ***and*** $\circledast \in \{+, *\}$ **then**
17 **return** $\texttt{Tree}(\circledast, \texttt{Operand}(\tau))$;
18 **else if** $\tau : \tau_1 \oplus \tau_2$ **then**
19 **return** $\texttt{Leaf}(\oplus, STT(\tau_1), STT(\tau_2))$;
20 **else**
21 **return** $\texttt{Leaf}(\tau)$;

- *STT* constructs a tree node if a term τ only uses addition or multiplication (lines 16–17). For other cases, *STT* creates a leaf node (lines 18–21).

Example 9. Fig. 7a and Fig. 7b show the parse trees of the symbolic representations for the data constraints in Fig. 1b and Fig. 1d, respectively. φ^* represents $\texttt{contains}(t.ty,$ 'IN'). Their isomorphism proves the data constraint equivalence.



(a) (b)

Fig. 7: Two isomorphic parse trees

It is worth noting that the AHU algorithm in Alg. 2 is slightly different from the standard one [13]. Originally, the AHU algorithm sorts the sub-trees by level, as the orders of the sub-trees do not matter. In our case, however, only the sub-trees of tree nodes can be arbitrarily permuted. Thus, we modify the AHU algorithm to adapt it to the isomorphism analysis, not sorting the sub-trees of each leaf node.

Algorithm 3: Decision procedure

Input: r_1, r_2: Two data constraints;
Output: Whether $r_1 \simeq r_2$ or not

1 $\varphi_1, \varphi_2 \leftarrow$ `getSymReps`(r_1, r_2);
2 **if** `Divergent`(φ_1, φ_2) *is true* **then**
3 **return** *false*;
4 **if** `Isomorphic`(φ_1, φ_2) *is true* **then**
5 **return** *true*;
6 **return** *(*`SMT-Solve`$(\neg(\varphi_1 \leftrightarrow \varphi_2))$ *is UNSAT)*;

C. Equivalence Verification with EQDAC

Combining the above analyses with the SMT solving, we obtain the decision procedure EQDAC in Alg. 3. We first construct the symbolic representations φ_1 and φ_2 via the semantic encoding. If we can not refute or prove the equivalence with the first two analyses (lines 2–5), an SMT solver examines whether φ_1 and φ_2 are logically equivalent (line 6) for general cases. Notably, the function *Divergent* invokes Alg. 1 at the line 2, and explores the Boolean structures of φ_1 and φ_2 at most two times, ensuring the efficiency of the first stage.

The divergence analysis and isomorphism analysis over and under-approximate the equivalence, respectively. Although the analyses do not always determine the equivalence, they can handle a large proportion of data constraints in practice, evidenced by our evaluation. Formally, we state two theorems to formulate the theoretical guarantee, of which the proofs are provided in [18].

Theorem 2. The steps in Alg. 3 before line 6 run in polynomial time to N, where N is the upper bound of the numbers of AST nodes for the two data constraints.

Theorem 3. For the syntax in Fig. 4, the data constraints are semantically equivalent if Alg. 3 returns true and vice versa.

VII. Implementation

We have implemented EQDAC in Python and deployed it in the FinTech company A. EQDAC first generates the AST of a data constraint and then translates it to the symbolic representation. We leverage the Z3 SMT solver [19] to support the SMT solving in the third stage. Particularly, we utilize the bit-vector, floating-point arithmetic, and string theory to encode variables and literals in the finite-length integer, floating point, and string types, respectively.

Based on EQDAC, we have further implemented two bots, which are shown in Fig. 2, to conduct the equivalence clustering and searching, respectively. In the equivalence clustering, we verify the equivalence of data constraints by invoking EQDAC pairwise. Particularly, we cache the symbolic representation of each data constraint to avoid redundant construction in different invocations. Similarly, we examine the equivalence of a new data constraint and each existing one sequentially in the equivalence searching, and also generate the symbolic representation for a data constraint only once.

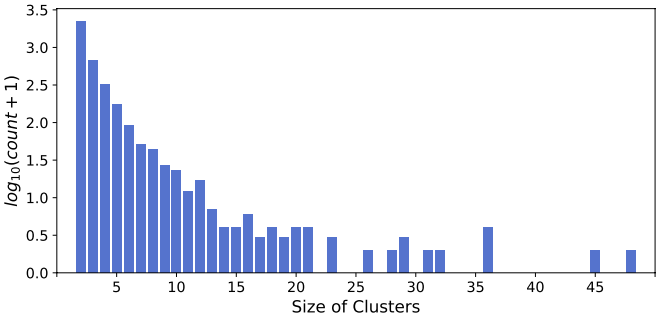


Fig. 8: The counts and sizes of clusters

VIII. Evaluation

To quantify the effectiveness and efficiency, we evaluate EQDAC upon the data constraints in a FinTech system by investigating the following research questions:

- **(RQ1)** How many equivalent data constraints are identified?
- **(RQ2)** How efficient is EQDAC in the equivalence clustering and searching?
- **(RQ3)** How important is each of the three stages?

Subjects. We collect 30,801 data constraints from a FinTech system in Company A, which are in the syntax shown in Fig. 4. Averagely, a data constraint contains 9.4 data constraints and 17.6 lines of code. Despite the moderate average size, we still need to handle the large set of data constraints efficiently, which is non-trivial yet crucial in industrial scenarios. Lastly, there are 1,497 data constraints not obeying our syntax, which are not selected as the subjects. They mainly contain advanced string operations, e.g., *substring* and *replaceAll*, and system calls, e.g., *getTimeZone*. In this work, EQDAC focuses on the data constraints in our given syntax, covering most of the data constraints (95.4%) in the FinTech system of Company A.

Environment. We conduct all the experiments on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20 GHz and 512 GB of physical memory. We invoke the Z3 SMT solver with its default options. We run the experiments with a time limit of 12 hours and a memory limit of 16 GB.

Availability. We release the code and sample constraints in GitHub repository [20]. The whole set of data constraints cannot be shared because of confidentiality agreements.

A. Equivalent Data Constraint Identification

To answer the first question, we evaluate EQDAC upon 30,801 data constraints by verifying the data constraint equivalence pairwise. Specifically, each pair of data constraints is fed to EQDAC to determine whether they are equivalent.

Result. We find that 11,538 data constraints (37.5%) have one or more equivalent variants, forming 26,789 equivalent pairs and 3,696 equivalence clusters. Particularly, we can leave one data constraint in each equivalence cluster and eliminate 7,842 data constraints without compromising the validity of the data reconciliation. Due to our limited permission, we sample a subset of the data constraints and measure the CPU time reduction when avoiding checking redundant ones. The

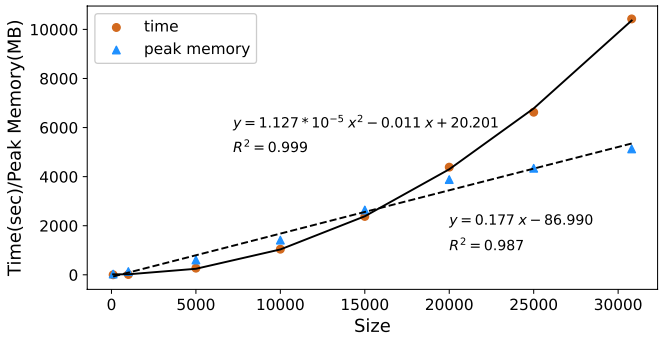


Fig. 9: Time and memory cost of equivalence clustering

result shows that the CPU time reduction ratio reaches 15.48%. According to the feedback of the experts, any reduction can bring a drastic benefit to the overall system in the long run, as data constraints are frequently checked online during a long development cycle.

We also count the data constraints in each cluster, in which data constraints are equivalent to each other. Fig. 8 shows that the size of a cluster ranges from 2 to 48. Specifically, the number of clusters with a size of 2 is 2,233. For the largest cluster with the size of 48, a violation of any data constraint will generate 48 alerts. Therefore, identifying equivalent data constraints can provide practical guidance in reusing the checking results and support the redundant alert elimination.

B. Performance Evaluation

We investigate the time consumption and memory usage of EQDAC in the equivalence clustering and searching. The experimental configurations are set up as follows.

- *Equivalence clustering*: To quantify the cost of clustering different sizes of data constraint sets, we construct eight sets of the data constraints, of which the sizes range from 100 to 30,801, and measure the time and memory usage of the clustering. All the data constraints are selected randomly.
- *Equivalence searching*: We select 1,000 data constraints from 30,801 data constraints as the recently-submitted ones and regard the remaining as the existing ones. Specifically, half of the selected ones are equivalent to at least one data constraint in the remaining set to quantify the cost of the equivalence searching in the worst case.

Result. As shown by Fig. 9, EQDAC finishes analyzing 30,801 data constraints in 2.89 hours within 5.01 GB of peak memory. We perform the regression analysis to quantify the scalability, choosing the quadratic and linear functions as the templates of the regression analyses for the time and memory cost, respectively, as we construct a symbolic representation for each data constraint only once and invoke the decision procedure in a pairwise manner. The R-squared values for memory and time are 0.987 and 0.999, respectively. Also, the coefficients in the quadratic and linear terms are quite small, indicating that the overhead increases gently. In summary, EQDAC supports the scalable equivalence clustering.

TABLE I: The statistics of the equivalence clustering

Variant	Time(h)	Mem(GB)	#Eq Pair	#Redundant
EQDAC-ND	OOT	7.27	141	53
EQDAC-NI	4.48	6.80	26,789	7,842
EQDAC-NS	2.13	3.94	25,952	7,296
EQDAC	2.89	5.01	26,789	7,842

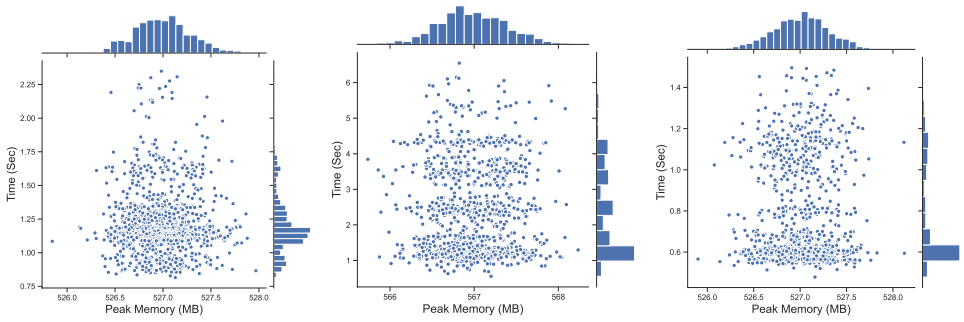
Fig. 10a shows the cost of the equivalence searching. All the analyses finish in 2.5 seconds within 528 MB of peak memory. Specifically, there is little difference in memory cost, ranging from 525.85 MB to 527.87 MB, while the time cost has a relatively large variance. The analyses of several data constraints demand SMT solving, which introduces more time costs. Typically, most of the cases can be analyzed in 1.5 seconds, and the average time cost is only 1.22 seconds. Thus, EQDAC supports searching equivalent data constraints efficiently, which is essential for maintenance.

C. Ablation Study

We set three ablations, namely EQDAC-ND, EQDAC-NI, and EQDAC-NS, which skip the divergence analysis, the isomorphism analysis, and SMT solving, respectively. The first two ablations are sound and complete, while EQDAC-NS can return *unknown* due to its incompleteness.

Result. Table I shows the results of the ablation study in the equivalence clustering. As we can see, EQDAC-ND does not finish analyzing 30,801 data constraints in 12 hours, and its peak memory reaches nearly 7.27 GB. Specifically, EQDAC-ND only finishes comparing seven data constraints with the remaining data constraints pairwise, discovering 141 equivalent pairs and 53 redundant data constraints. EQDAC-NI discovers the same equivalent pairs as EQDAC. However, it has to perform the SMT solving for all the data constraint pairs of which the equivalence is not refuted by the divergence analysis, increasing the time cost to 4.48 hours. EQDAC-NS skips the SMT solving and consumes less time and memory than EQDAC, spending 1.78 and 0.35 hours on the divergence analysis and the isomorphism analysis, respectively. It does not discover 837 equivalent pairs, and thus, misses 546 redundant constraints. Particularly, our divergence analysis identifies 38,964 non-equivalent pairs even if their symbolic representations have the same sets of data variables, literals, and operators. Thus, the divergence analysis not only refutes the equivalence soundly but also provides the possibility of refuting more non-equivalent pairs.

Fig. 10 shows the cost of the equivalence searching. EQDAC-NI costs more in each equivalence searching task, as the SMT solver consumes more resources to prove the equivalence. Specifically, its average time cost is 2.53 seconds, and its peak memory reaches 566.98 MB. In the worst case, it takes 6.56 seconds to finish the equivalence searching of a data constraint, degrading its usability in real-world production. EQDAC-NS consumes less time because it does not invoke SMT solvers in all the cases. However, it can not identify the equivalent variants for 37 data constraints due to incompleteness. EQDAC-ND does not finish the equivalence



(a) (b) (c)

Fig. 10: Time and memory cost of EQDAC, EQDAC-NI, and EQDAC-NS

```
/* Data contraint 1 */
assert(t.id != t.pid);
assert(ut.oid != ut.iid);
if(t.id == ut.oid){
  assert(t.pid == ut.iid);
} else {
  assert(t.id == ut.iid);
  assert(t.pid == ut.oid);}

/* Data contraint 2 */
if(t.id == ut.iid){
  assert(ut.oid == t.pid);
} else {
  assert(t.id == ut.oid);
  assert(t.pid == ut.iid);}
assert(ut.iid != ut.oid);
assert(t.pid != t.id);
```

Fig. 11: An example of case study

searching of 1,000 data constraints in the given time budget. It has to invoke the SMT solver to prove the non-equivalence, making the overall time cost unacceptable.

Case Study. Fig. 11 shows an equivalent pair discovered via the SMT solving. The data constraints both examine whether the IDs of the income and expense accounts match with the ones in the transaction. Unfortunately, we can not deduce the equivalence from the parse trees of their symbolic representations. Instead, we have to reason multiple assertions in a relational manner. The two assertions in the sequencing are the premise of the equivalence of two $\textbf{ite}_s$ statements, while it is beyond the ability of the isomorphism analysis.

D. Discussion

In what follows, we demonstrate the discussions on the feedback from the users, threats to validity, limitations, and future work.

Feedback from the Users. EQDAC has been integrated into the production line of Company A, serving as the core building block of two bots in the CI/CD workflow. To obtain the feedback of users, we assigned the questionnaires to the developers and the quality assurance managers in the forum of the company, which received rave reviews from users. For example, a developer comments the search bot in the forum as follows, showing his appreciation for the instant response and useful results.

> *"The search is so smooth! I had been expecting such an assistant for data constraint maintenance. The results are mostly fetched in just one or two seconds, assisting in merging data constraints."*

Threats to Validity. A threat to validity is whether the way of producing data constraints affects our results. Ineffective communication between developers could increase the number of equivalent data constraints, as they are unaware of the data constraints submitted by others. For a small FinTech system with only a few data constraints, the benefit of resolving redundancy could be less significant. EQDAC mainly targets systems with thousands of data constraints and shows excellent potential to improve their maintainability.

Limitations and Future Work. First, our syntax excludes several string operations. Theoretically, solving general string constraints is undecidable [21]–[23], while the first two stages of EQDAC can still work in the presence of advanced string operations. It would be interesting to reason more string operations even though, according to our experience, they do not widely exist. Second, EQDAC focuses on equivalence relations in this work. It is meaningful to examine whether a data constraint subsumes others for consolidation [24]. Third, EQDAC currently analyzes data constraints in FinTech systems. It would be promising to extend EQDAC to support light-weighted equivalence checking in other domains, such as SQL queries [9] and database-backed programs [25].

IX. Related Work

A. Program Equivalence Checking

There is an extensive body of research on program equivalence checking, which is a crucial building block in many clients, such as translation validation [7], [26] and program synthesis [27]–[29]. One line of studies reduces equivalence checking to proving specific verification conditions, such as *relational verification* [30]–[35]. Similar approaches include using symbolic execution for loop-free programs [28], [36]–[39]. Different from the existing efforts that target sophisticated program constructs, EQDAC focuses on a domain-specific language in FinTech systems, pursuing efficiency over the capability of handling a flexible program syntax.

Another line of studies proves program equivalence via *term rewriting* [6], [7], [40], [41]. The effectiveness relies heavily on the quality of rewrite rules. First, they may sacrifice soundness or completeness if the rule set contains an incorrect rule or misses a right one [42]. Second, they may suffer from the phase ordering problem [8] in the presence of a large number of rewrite rules. To obtain better complexity, [43] restricts the form of rewrite rules, and adopts tree isomorphism algorithms to check syntactic isomorphism. EQDAC bears similarities to [43] in terms of proving the equivalence, while we consider more program constructs in the isomorphism analysis, such as arithmetic operators and string predicates, which promotes its capability in practice.

B. SQL Query Equivalence Checking

Verifying SQL query equivalence is an essential topic in both academia and industrial communities. The state-of-the-art approaches focus on specific forms of SQL queries [44] and apply either algebraic reasoning techniques [6], [41], [45] or symbolic reasoning [9], [10], [46] for equivalence verification. Typically, UDP [41] utilizes U-semiring to encode the bag semantics of SQL effectively and checks the isomorphism between two algebraic structures. However, it fails to handle advanced features, e.g., three-valued logic, and suffers from the inefficient chase procedure in the isomorphism checking [47]. In contrast, EQUITAS [9] encodes the semantics with a FOL formula and leverages a solver to determine the equivalence, handling more SQL features [48] than UDP.

In our work, EQDAC targets the equivalence of data constraints rather than SQL queries, while it bears similarities with existing efforts in terms of technical designs. Specifically, it abstracts away the orders of commutative constructs, and leverages a solver to determine the equivalence of two FOL formulas. Thus, it avoids the inefficient chase procedure and unleashes the power of SMT solving.

C. SMT Solving Optimization

There is a vast amount of literature on guiding SMT solving with program features. One typical line of studies performs semantically equivalent transformations to reduce the overhead of SMT solving [12], [49]–[52], For example, [51] uses contextual information to simplify array constraints, which transforms array operations with symbolic indexes to the ones only involving constant indexes. Another line of the literature utilizes semantic information to optimize SMT solving algorithms [53]–[56]. For instance, [52], [53] leverage control-flow information to guide the branching strategy in CDCL(T)-style SMT solving. Similarly, our symbolic representations preserve program syntactic features, which supports proving the equivalence efficiently by the isomorphism analysis.

Apart from accelerating the solving process, caching the intermediate results, e.g., unsatisfiable core [57] and syntactic features [58], can also avoid calling the solver. GREEN examines the syntactic equivalence of constraints, and reuses the previous solving result of the equivalent ones [58]. It shares a similar idea with the isomorphism analysis of EQDAC, while EQDAC also conducts the divergence analysis to discover non-equivalent pairs efficiently, promoting the efficiency of data constraint maintenance applications in the real world.

X. CONCLUSION

We have presented EQDAC, an efficient, sound, and complete decision procedure for verifying the data constraint equivalence in FinTech systems. It supports two typical clients, namely equivalence clustering and searching, in the production line of a global FinTech company. EQDAC scales to a large number of data constraints with high efficiency, liberating productivity in the data constraint maintenance. We believe that the insight behind EQDAC can further promote equivalence checking in other domains.

ACKNOWLEDGMENT

We thank the anonymous reviewers for valuable feedback on earlier drafts of this paper, which helped improve its presentation. We also appreciate Dr. Xiao Xiao for insightful discussions. The authors are supported by the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei. Peisen Yao is the corresponding author.

REFERENCES

[1] Feifei Li. Cloud native database systems at alibaba: Opportunities and challenges. *Proc. VLDB Endow.*, 12(12):2263–2272, 2019.

[2] Juan Manuel Florez, Laura Moreno, Zenong Zhang, Shiyi Wei, and Andrian Marcus. An empirical study of data constraint implementations in java. *Empir. Softw. Eng.*, 27(5):119, 2022.

[3] Juan Manuel Florez, Jonathan Perry, Shiyi Wei, and Andrian Marcus. Retrieving data constraint implementations using fine-grained code patterns. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1893–1905. ACM, 2022.

[4] Neil Ernst, Rick Kazman, and Julien Delange. *Technical Debt in Practice: How to Find It and Fix It*. MIT Press, 2021.

[5] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25:1–25:45, 2013.

[6] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 510–524. ACM, 2017.

[7] George C. Necula. Translation validation for an optimizing compiler. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 83–94. ACM, 2000.

[8] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[9] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. Automated verification of query equivalence using satisfiability modulo theories. *Proc. VLDB Endow.*, 12(11):1276–1288, 2019.

[10] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. A symbolic approach to proving query equivalence under bag semantics. *arXiv preprint arXiv:2004.00481*, 2020.

[11] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 13–24. ACM, 2020.

[12] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Daniel Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. Block public access: trust safety verification of access control policies. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 281–291. ACM, 2020.

[13] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[14] Abdelghani Bakhtouchi. Data reconciliation and fusion methods: A survey. *Applied Computing and Informatics*, 2020.

[15] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*, pages 471–482. IEEE, 2021.

[16] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1098–1109. ACM, 2020.

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[18] Chengpeng Wang, Gang Fan, Peisen Yao, Fuxiong Pan, and Charles Zhang. Verifying data constraint equivalence in fintech systems. *CoRR*, abs/2301.11011, 2023.

[19] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[20] EqDAC. Equivalence Verification of Data Constraints. https://github.com/EqDAC/EqDACTool, 2022. [Online; accessed 7-Sept-2022].

[21] Taolue Chen, Yan Chen, Matthew Hague, Anthony W Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[22] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, 2012.

[23] Taolue Chen, Matthew Hague, Anthony W Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[24] Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. Consolidation of queries with user-defined functions. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 554–564. ACM, 2014.

[25] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. Extracting equivalent SQL from imperative code in database applications. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1781–1796. ACM, 2016.

[26] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482. ACM, 2013.

[27] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 305–316. ACM, 2013.

[28] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. Synthesizing replacement classes. *Proc. ACM Program. Lang.*, 4(POPL):52:1–52:33, 2020.

[29] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 31–44. ACM, 2020.

[30] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. Client-specific equivalence checking. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 441–451. ACM, 2018.

[31] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. Modular demand-driven analysis of semantic difference for program versions. In *International Static Analysis Symposium*, pages 405–427. Springer, 2017.

[32] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. Constraint-based relational verification. In *International Conference on Computer Aided Verification*, pages 742–766. Springer, 2021.

[33] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 57–69. ACM, 2016.

[34] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1027–1040. ACM, 2019.

[35] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. Differential symbolic execution. In Mary Jean Harrold and Gail C. Murphy, editors, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 226–237. ACM, 2008.

[37] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*, pages 712–717. Springer, 2012.

[38] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 13–24, 2020.

[39] Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. Relational verification using reinforcement learning. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[40] Benjamin Goldberg, Lenore D. Zuck, and Clark W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electron. Notes Theor. Comput. Sci.*, 132(1):53–71, 2005.

[41] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endow.*, 11(11):1482–1495, 2018.

[42] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021.

[43] Simon Guilloud and Viktor Kuncak. Equivalence checking for ortho-complemented bisemilattices in log-linear time. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 196–214. Springer, 2022.

[44] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 205–213. ACM Press, 1998.

[45] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. Demonstration of the cosette automated SQL prover. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1591–1594. ACM, 2017.

[46] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Speeding up symbolic reasoning for relational queries. *Proc. ACM Program. Lang.*, 2(OOPSLA):157:1–157:25, 2018.

[47] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 207–224. Springer, 2003.

[48] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. SIA: optimizing queries using learned predicates. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2169–2181. ACM, 2021.

[49] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 651–664. ACM, 2021.

[50] Cristian Cadar. Targeted program transformations for symbolic execution. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 906–909. ACM, 2015.

[51] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 68–78. ACM, 2017.

[52] Jianhui Chen and Fei He. Leveraging control flow knowledge in SMT solving of program verification. *ACM Trans. Softw. Eng. Methodol.*, 30(4):41:1–41:26, 2021.

[53] Jianhui Chen and Fei He. Control flow-guided SMT solving for program verification. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 351–361. ACM, 2018.

[54] Hongyu Fan, Weiting Liu, and Fei He. Interference relation-guided SMT solving for multi-threaded program verification. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 163–176. ACM, 2022.

[55] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 930–943, 2021.

[56] Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. Type and interval aware array constraint solving for symbolic execution. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 361–373, 2021.

[57] Jan Mrázek, Martin Jonás, Vladimír Still, Henrich Lauko, and Jiri Barnat. Optimizing and caching SMT queries in symdivine - (competition contribution). In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 390–393, 2017.

[58] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 58. ACM, 2012.