EUF^n: A Decidable Extension to the Theory of Equality with Uninterpreted Functions

YIDE DU[†], National University of Defense Technology, China
ZHENBANG CHEN[*†], National University of Defense Technology, China
WEIJIANG HONG[†], National University of Defense Technology, China
WEI DONG[†], National University of Defense Technology, China

The theory of Equality with Uninterpreted Functions (EUF) is fundamental to constraint solving and program verification. Uninterpreted functions abstract concrete implementations, enabling generalization and simplification of theorems and proofs. However, standard EUF restricts function composition to fixed finite depths (*e.g.*, $f^k(x)$ where k is constant). This work extends EUF to EUF^n, supporting *parametric composition depth* for unary functions (*e.g.*, $f^n(x)$ where n is a natural number variable).

An EUF^n formula can be viewed as a disjunction of infinitely many EUF formulas, each instantiated by an assignment of natural numbers. Its satisfiability is defined by the satisfiability of at least one such instantiated EUF formula. We establish the decidability of the EUF^n satisfiability problem via a *conditional congruence graph (CCG)* algorithm. This approach generalizes the standard congruence closure procedure by maintaining conditional equivalence relations between terms. The algorithm reduces the satisfiability problem to deciding existential sentences in Presburger arithmetic with divisibility, which is a decidable problem, thereby yielding a decision procedure for the quantifier-free fragment of EUF^n with a 2NEXPTIME complexity upper bound.

The enhanced expressiveness of EUF^n enables new applications: (1) Encoding a decidable subclass of interleaved Dyck reachability problems where existing over/under-approximations produce false positives/negatives, and (2) Encoding a new decidable subclass of uninterpreted program verification problems.

CCS Concepts: • **Theory of computation → Logic and verification**.

Additional Key Words and Phrases: EUF, Decidable Extension, Interleaved Dyck Reachability, Program Verification

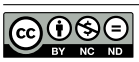
ACM Reference Format:
Yide Du, Zhenbang Chen, Weijiang Hong, and Wei Dong. 2026. EUF^n: A Decidable Extension to the Theory of Equality with Uninterpreted Functions. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 329 (October 2026), 29 pages. https://doi.org/10.1145/3839461

1 Introduction

The theory of Equality with Uninterpreted Functions (EUF) establishes a logical framework combining equality reasoning with uninterpreted function symbols, where variables range over infinite

[*]Zhenbang Chen is the corresponding author.
[†]Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China.

Authors' Contact Information: Yide Du, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China, dyd1024@nudt.edu.cn; Zhenbang Chen, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China, zbchen@nudt.edu.cn; Weijiang Hong, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China, hongweijiang17@nudt.edu.cn; Wei Dong, College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, China, wdong@nudt.edu.cn.



ACM 2475-1421/2026/10-ART329
https://doi.org/10.1145/3839461

domains and functions retain only their *functional congruence* property. Formally, this congruence axiom requires that identical inputs produce identical outputs: $\forall x, y.\ x = y \rightarrow f(x) = f(y)$. A canonical demonstration of this principle appears in the EUF formula: $x = y \wedge f(x) \neq f(y)$ which is inherently unsatisfiable due to the contradiction between the assumed equality $x = y$ and the required congruence consequence $f(x) = f(y)$.

As a fundamental component of Satisfiability Modulo Theories (SMT), EUF plays a pivotal role in formal verification methodologies [Bueno 2021]. Its applications encompass a wide range of areas, including circuit equivalence verification [Burch and Dill 1994], the proof of program equivalence before and after compiler optimizations [Müller-Olm et al. 2005], verification of uninterpreted programs [Govind et al. 2021], and model checking [Fan and He 2024; Lee and Sakallah 2014]. The essential power of EUF lies in its abstraction mechanism: uninterpreted functions obscure implementation details to simplify verification. When verification fails under this abstraction, practitioners employ a *refinement process* that incrementally incorporates specific function properties until either identifying genuine counterexamples or completing the proof. This methodology strikes an optimal balance between abstraction precision and verification tractability.

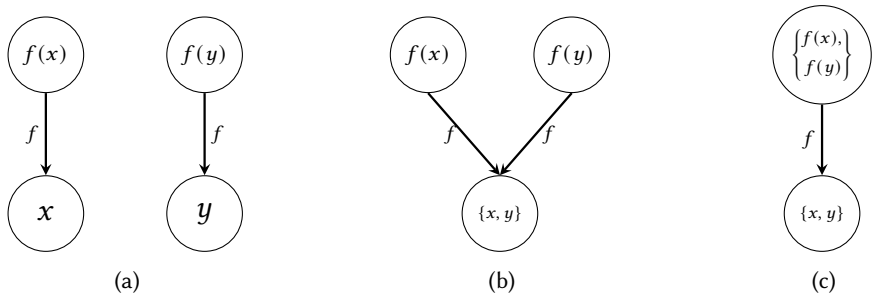


Fig. 1. Congruence Closure on an EUF Example.

The satisfiability problem for EUF formulas is known to be decidable. The foundational work by [Shostak 1984], [Nelson and Oppen 1980], and [Downey et al. 1980] established the congruence closure algorithm, which combines directed acyclic graph (DAG) representations with the *Union-Find* data structure to maintain distinct equivalence classes. Here, $Union(x, y)$ merges the equivalence classes containing x and y while designating a new representative, and $Find(x)$ retrieves the representative of the equivalence class containing x. To illustrate, consider the prototypical EUF formula:

$$x = y \wedge f(x) \neq f(y).$$

As illustrated in Figure 1a, the initial equivalence classes are formed as distinct sets: $\{x\}$, $\{y\}$, $\{f(x)\}$, and $\{f(y)\}$, which correspond to the individual nodes in the graph. The directed edges represent the superterm relationship between nodes; for example, $f(x)$ is the term obtained by applying the function f to x, and it is referred to as a superterm of x. The algorithm then processes the equalities in the formula. Upon processing $x = y$, $Union(x, y)$ is called to merge the equivalence classes of x and y into $\{x, y\}$, as shown in Figure 1b. This merger establishes $f(x)$ as a superterm of y and $f(y)$ as a superterm of x. Consequently, congruence propagation is triggered: since x and y are now equivalent, functional congruence requires that $f(x)$ and $f(y)$ must also be equivalent, leading to the $Union(f(x), f(y))$ operation (Figure 1c). Finally, when processing the disequality $f(x) \neq f(y)$, the condition $Find(f(x)) = Find(f(y))$ indicates that $f(x)$ and $f(y)$ belong to the same equivalence class, directly contradicting the disequality. This conflict, detected via congruence closure, confirms the formula's unsatisfiability.

A fundamental limitation in standard EUF theory is its support for only *finite-depth function composition*, as seen in terms like $f^c(x)$, where c is a constant (*e.g.*, $f(f(x))$ and $f^6(x)$). To enhance the expressiveness of standard EUF, we propose EUF^n—a decidable extension that allows *parametric composition depth* for unary functions through terms of the form $f^n(x)$, where n is a natural number variable. The essential distinction lies in the finite notation $f^n(x)$, which has no direct analogue for multi-arity functions. Consequently, our extension must prohibit parametrically nested compositions in the latter case. For example, a term such as $g(a, g(b, g(c, d)))$ has no concise parametric representation like $g^3(\cdot, \cdot)$. On the other hand, terms that mix parametric and fixed-depth applications—such as $g(f^n(x), x)$ or $g(f^2(x), x)$—remain well-formed and admissible within EUF^n. Despite this restriction, EUF^n remains sufficiently expressive for many verification tasks involving unary function abstractions. For instance, it can model data-structure traversals—*e.g.*, $read^n(arr)$ for array access, $next^n(head)$ for list walking—where the parametric depth n captures the access index or path length. It is also applicable to control-property verification: by using a unary counting function $add^n(count)$, one can encode constraints such as equality or inequality between the execution counts of two functions, which is useful for detecting anomalies like use-after-free. On the other hand, the term $f^n(x)$ can be expressed in first-order logic as $f'(n, x)$, where n is a natural number variable and x is an uninterpreted variable. However, since the current combination of EUF and Presburger arithmetic does not support the derivation of equalities such as $f'(1, f'(n, x)) = f'(n + 1, x)$, it is necessary to extend the syntax and axioms of EUF and to develop specialized algorithms for solving EUF^n formulas.

This paper introduces the syntax and axioms of EUF^n, along with a *conditional congruence graph* (*CCG*) algorithm for solving quantifier-free EUF^n formulas, establishing the decidability of their satisfiability problem. Analogous to the standard congruence closure algorithm for EUF (Figure 1), this approach represents terms as nodes and functional relationships as edges. The key distinction lies in the *CCG*, which maintains both conditional equivalence and conditional superterm relationships, explicitly capturing the conditions under which terms become equivalent or functionally related. For example, a conditional equivalence edge between terms $f^m(x)$ and $f^n(x)$ is annotated with the condition $m{=}n$. As a result, when processing equalities in EUF^n formulas, the algorithm updates only the associated conditional constraints, rather than merging the corresponding nodes as in Figure 1. Finally, the algorithm reduces satisfiability of EUF^n formulas to the satisfiability problem for the existential theory of Presburger arithmetic with divisibility, which is known to be decidable [Bel'tyukov 1980; Lipshitz 1978] in NEXPTIME [Lechner et al. 2015] thereby establishing decidability.

EUF^n extends the expressiveness of standard EUF, enabling broader applications. On one hand, the congruence property of functions closely parallels bracket matching: terms obtained by applying the same function the same number of times are treated as equivalent, analogous to matched brackets in balanced expressions. This correspondence allows EUF^n to encode Dyck reachability problems over graphs, where two nodes are Dyck-reachable if a path between them forms a well-balanced bracket string in a Dyck language (*i.e.*, matching brackets). Dyck reachability [Zhang et al. 2013] and its extension, interleaved Dyck reachability [Conrado and Pavlogiannis 2024; Li et al. 2023; Zhang and Su 2017], play a central role in static analysis with wide-ranging applications. Although interleaved Dyck reachability is not an equivalence relation and is generally undecidable [Reps 2000], we show that in the single-source single-target setting, a decidable subclass can be encoded using EUF^n. In contrast, existing over- and under-approximation techniques may still yield false positives or negatives even in this restricted case.

Additionally, uninterpreted functions abstract concrete implementations to simplify verification. Programs that use such abstractions, which are referred to as uninterpreted programs, have undecidable verification in general [Mathur et al. 2019a]. Unlike prior approaches that reason over

program states (*e.g.*, equalities, disequalities and function relations) [Govind et al. 2021; Hong et al. 2021; Mathur et al. 2019a], EUF^n supports encoding of certain *execution traces*. For instance, a loop like `while(cond){x=next(x);}` yields access chains $next^n(x)$, where n depends on iteration count. Standard EUF loses this parametric relation by requiring concrete n [Mathur et al. 2019b], while EUF^n retains it symbolically, which identifies a decidable subclass of uninterpreted programs beyond the *coherent* class [Govind et al. 2021; Mathur et al. 2019a].

To summarize, this paper presents the following contributions:

(1) We propose EUF^n, an extension of EUF that enables the representation of terms with *parametric composition depth* for unary functions.
(2) We establish the decidability of the quantifier-free fragment of EUF^n via the *CCG* algorithm with 2NEXPTIME complexity.
(3) We demonstrate the practical expressiveness of EUF^n by encoding a subclass of single-source single-target interleaved Dyck reachability problems, as well as a subclass of verification problems for uninterpreted programs. In both cases, we identify new decidable subclasses enabled by this encoding.

The structure of this paper is organized as follows: Section 2 introduces the basic concepts of EUF^n, Section 3 provides the decision algorithm for EUF^n, Section 4 examines the practical encoding capabilities of EUF^n, Section 5 reviews related work, and Section 6 concludes the paper.

2 Logic of EUF^n

This section presents the formal syntax and axioms of EUF^n, followed by a detailed example illustrating its decision procedure.

2.1 Syntax of EUF^n

The syntax of EUF^n is defined over a finite signature $\Sigma = \mathsf{V} \cup \mathsf{F} \cup \mathsf{C}$. Variables are partitioned into $\mathsf{V} = \mathsf{U} \cup \mathsf{N}$, where U consists of uninterpreted-type variables ($var \in \mathsf{U}$) and N includes natural number-type variables ($int\_var \in \mathsf{N}$). Here, F denotes the set of function symbols, which includes unary function symbols (*e.g.*, the successor function S) and k-ary function symbols for $k \geq 2$ (*e.g.*, the binary symbol +). A composite unary term can be written as $f_1 \circ \cdots \circ f_n$, where each $f_i \in F$ $(1 \leq i \leq n)$ is a unary function symbol. Finally, C denotes the set of integer constants, with $0, c, c_1, c_n \in C$.

$$
\begin{aligned}
formula &::= \textbf{true} \mid \textbf{false} \mid term = term \\
&\quad \mid formula \wedge formula \mid formula \vee formula \mid \neg formula \\
term &::= var \mid f^{lin}(term) \mid g(term, \cdots, term) \\
lin &::= c \mid int\_var \mid c_1 \cdot lin + \cdots + c_n \cdot lin
\end{aligned}
$$

Fig. 2. The syntax of EUF^n formulas.

As formalized in Figure 2, the syntax employs a three-tiered structure:

- $formula$ represents an EUF^n formula, constructed from Boolean literals (**true**, **false**), equality predicates over $term$s (atomic formulas), or logical combinations of subformulas using standard connectives.
- $term$ denotes functional expressions within formulas, which may be uninterpreted-type variables (var) or function applications, where f represents a unary function and g denotes a multi-arity function.

- *lin* specifies the composition depth for unary function f, defined as either constants (c), natural number-type variables (*int_var*), or a linear combination of existing *lin* expressions. The value of any *lin* expression is guaranteed to be non-negative during formula solving.

2.2 The Axioms of EUF^n

To enable *parametric composition depth* for unary functions, we extend the EUF theory into a many-sorted EUF^n with two distinct sorts: U (uninterpreted) and N (natural numbers). The extended theory preserves the four core axioms of EUF, with $\vec{x}, \vec{y} \in \mathsf{U}^k$ denoting k-tuples:

(1) $\forall x : \mathsf{U}.\ x = x$ (reflexivity)
(2) $\forall x, y : \mathsf{U}.\ x = y \rightarrow y = x$ (symmetry)
(3) $\forall x, y, z : \mathsf{U}.\ x = y \wedge y = z \rightarrow x = z$ (transitivity)
(4) $\forall \vec{x}, \vec{y} : \mathsf{U}^k.\ \bigwedge_{i=1}^{k} x_i = y_i \rightarrow f(\vec{x}) = f(\vec{y})$ (congruence)

This extension further requires the axioms of Presburger arithmetic (see Appendix A) along with the following two novel axioms governing function composition depth:

(5) $\forall x : \mathsf{U}.\ f^0(x) = x$
(6) $\forall x : \mathsf{U}, n : \mathsf{N}.\ f^{S(n)}(x) = f(f^n(x))$

where x is a variable of uninterpreted type (U) and n is a natural number variable of type (N), while S denotes the successor function.

The model of EUF^n is defined as $\mathcal{M} = (\mathcal{U} \cup \mathbb{N}, \mathcal{I})$, where $\mathcal{U}$ denotes the domain of uninterpreted elements, $\mathbb{N}$ denotes the domain of natural numbers. For every unary function $f : \mathsf{U} \rightarrow \mathsf{U}$, there exists a corresponding binary function $f' : \mathsf{N} \times \mathsf{U} \rightarrow \mathsf{U}$ such that for any variable $n : \mathsf{N}$, $x : \mathsf{U}$, and assignment functions $\sigma : \mathsf{N} \rightarrow \mathbb{N}$ and $\sigma' : \mathsf{U} \rightarrow \mathcal{U}$, we have

$$\mathcal{I}(f')(\sigma(n), \sigma'(x)) = \underbrace{\mathcal{I}(f) \circ \cdots \circ \mathcal{I}(f)}_{\sigma(n) \text{ times}}(\sigma'(x)).$$

A natural way to satisfy this requirement is for $\mathcal{I}(f')$ to apply $\mathcal{I}(f)$ repeatedly n times. In the axioms, we use $f^n(x)$ as a shorthand for $f'(n, x)$. The interpretations of addition (+), and constants in Presburger arithmetic follow the standard semantics of natural number addition theory.

Theorem 1 (Generalized Congruence).

$$\forall x, y : \mathsf{U},\ m, n : \mathsf{N}.\ x = y \wedge m = n \rightarrow f^n(x) = f^m(y)$$

Proof. The case where $m = n = 0$ is secured by Axiom (5). The general case can be established by induction, where the inductive step would rely on Axioms (6) and (4), thus allowing for the completion of the proof. □

Having defined the syntax and axioms of EUF^n formulas, we now define their satisfiability in terms of the satisfiability of EUF formulas [Shostak 1978]. For a given formula Ψ and an assignment $\sigma : \mathsf{N} \rightarrow \mathbb{N}$, we define $\Psi[\sigma]$ as the formula obtained by replacing every free occurrence of a variable x in Ψ with the corresponding value $\sigma(x)$.

Definition 1 (EUF^n Satisfiability). A formula Ψ in EUF^n over signature Σ is satisfiable iff there exists an assignment $\sigma : \mathsf{N} \rightarrow \mathbb{N}$ such that $\Psi[\sigma]$ is satisfiable in EUF.

2.3 An Example of the Decision Procedure for EUF^n

EUF^n extends EUF by supporting terms with *parametric composition depth* for unary functions, thereby transforming equivalence relations between terms into *conditional equivalence*: two terms become equivalent only when their composition depth satisfys specific constraints. To demonstrate

the decision procedure for EUF^n, we employ a concrete example to illustrate its core mechanisms. Consider the following EUF^n formula as a representative case study:

$$x = y \wedge z = f^m(x) \wedge k = f^n(y) \wedge z \neq k, \tag{1}$$

where $m, n \in \mathsf{N}$ denote the composition depth of the uninterpreted unary function f. This formula can be regarded as the disjunction of EUF formulas generated by instantiating m and n over the natural numbers $\mathbb{N}$, as shown below:

$$\begin{aligned} &x = y \wedge z = f^m(x) \wedge k = f^n(y) \wedge z \neq k \iff (x = y \wedge z = f^0(x) \wedge k = f^0(y) \wedge z \neq k) \vee \\ &(x = y \wedge z = f^1(x) \wedge k = f^0(y) \wedge z \neq k) \vee (x = y \wedge z = f^0(x) \wedge k = f^1(y) \wedge z \neq k) \vee \cdots \end{aligned} \tag{2}$$

The EUF^n formula is satisfiable if and only if there exists an assignment σ such that the corresponding EUF formula $x = y \wedge z = f^{\sigma(m)}(x) \wedge k = f^{\sigma(n)}(y) \wedge z \neq k$ is satisfiable.

Figure 3 presents the solving process of the formula based on the *conditional congruence graph* (*CCG*) algorithm. In these graphs, nodes represent *term*s from the formula, and directed edges indicate superterm relationships. For example, the *term* $f^m(x)$ (*i.e.*, z) is a superterm of x, with edge label $(m{>}0, f, m)$ encoding the function f, its composition depth m, and the condition $m > 0$ that validates the superterm relation. Undirected edges connect equivalent *term*s, annotated with logical conditions governing their equivalence. Figure 3a shows the initialized *CCG* for the example, which includes the conditional superterm relationships between the terms in the formula. Notably, during resolution, the graph preserves all existing nodes and introduces only new edges.

The *CCG* provides a symbolic representation of the family of concrete congruence graphs induced by the disjunctive EUF formulas above. For any concrete assignment to parameters m and n, the corresponding concrete congruence graph is obtained from the *CCG* via a two-step instantiation: first, evaluating the parametric conditions on its edges with the given values; second, keeping only those edges whose conditions are satisfied. After merging equivalent nodes, the resulting graph coincides exactly with the congruence graph of the instantiated EUF formula.

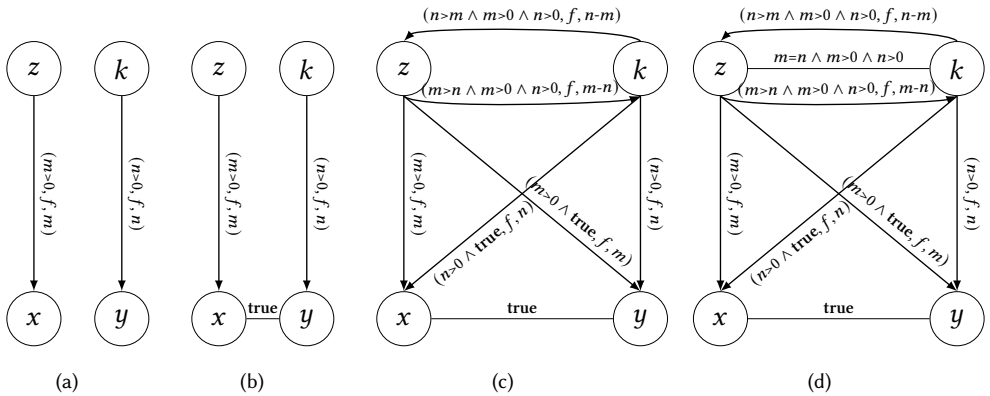


Fig. 3. *CCG*-Based Solution of an EUF^n Formula Example

The decision procedure starts by processing the equation $x = y$, which introduces an undirected edge between nodes x and y, annotated with the constraint **true**. To maintain correctness after inserting a conditional equivalence edge, three operations are performed on the *CCG*; collectively, they symbolically abstract the corresponding operations needed for the entire family of concrete congruence graphs it represents.

(1) **Equivalence propagation.** Let $[x]$ denote the set containing x and all nodes connected to x via conditionally equivalent edges (similarly for $[y]$). This operation corresponds to the *Union* procedure in standard EUF congruence closure algorithms. The step establishes pairwise

equivalence between all nodes in $[x]$ and $[y]$, updating their conditions by conjoining with the equivalence condition of x and y. This operation is illustrated in Figure 3b with the addition of the edge $x \xrightarrow{\textbf{true}} y$.

(2) **Superterm inheritance.** After adding the conditional equivalence between x and y, their superterms/subterms consequently become shared under the given condition. Therefore, in this step, x and y inherit each other's superterm/subterm relations and update the associated conditions. In Figure 3c, edges $k \xrightarrow{(n>0\wedge\textbf{true},f,n)} x$ and $z \xrightarrow{(m>0\wedge\textbf{true},f,m)} y$ are added. Moreover, new conditional superterm relationships can arise between terms such as $f^m(x)$ (*i.e.* z) and $f^n(y)$ (*i.e.* k). This follows from the extended formalism's support for parametric composition depth, which forces consideration of all possible functional relationships between terms. As shown in Figure 3c, the edges $z \xrightarrow{(m>n\wedge m>0\wedge n>0,f,m-n)} k$ and $k \xrightarrow{(n>m\wedge m>0\wedge n>0,f,n-m)} z$ are introduced.

(3) **Conditional congruence closure.** When solving EUF formulas, after adding equivalence relations, it is necessary to check for additional derivable equivalences based on congruence axioms. Similarly, when using the *CCG* to handle EUF^n formulas, conditional equivalence relations must also be processed to ensure (conditional) congruence closure. In Figure 3d, given the superterm relations $z = f^m(x)$ and $k = f^n(y)$, the edge $z \xrightarrow{m=n\wedge m>0\wedge n>0} k$ is added to the graph. In this case, there is no superterm relationship between x and y; however, when such a relationship exists, the congruence closure process differs. This will be discussed in detail in Section 3.5.

After adding the conditional equivalence between z and k, the same three steps repeat. Since z and k have no additional equivalent nodes (this simplified example omits edges such as the one from x to z with the constraint $m = 0$), **equivalence propagation** is unnecessary. The only conditional superterm of k is z, but the superterm condition $m = n \wedge m > n \wedge m < n$ is unsatisfiable, terminating **superterm inheritance** for z (and similarly for k). Finally, while z and k are mutual superterms, the superterm conditions remain unsatisfiable, so the **conditional congruence closure** terminates.

Finally, the decision procedure handles the disequality $z \neq k$ by examining the conditional equivalence relation between z and k in the *CCG*, subject to the constraint $m = n \wedge m > 0 \wedge n > 0$. Since $\neg(m = n \wedge m > 0 \wedge n > 0)$ is satisfiable (*e.g.*, by assigning distinct positive integers to m and n), it follows that in this case, no contradiction arises. Consequently, the original EUF^n formula is satisfiable for such assignments.

As shown in this example, for any assignment σ, substituting $\sigma(m)$ and $\sigma(n)$ for m and n in both the terms and edge conditions of Figure 3 determines which edges remain valid. The resulting congruence graph, which is obtained by merging equivalence edges with **true** conditions and removing redundant superterm edges, matches the congruence graph obtained by directly evaluating the EUF formula from Equation 2 under σ. Thus, the *CCG* in Figure 3 precisely captures the congruence graphs of all the disjuncted EUF formulas: x and y are always equivalent, while z and k are equivalent if and only if $\sigma(m) = \sigma(n)$.

3 The EUF^n Decision Algorithm

This section first presents a formal definition of the *conditional congruence graph (CCG)* and then describes the EUF^n decision algorithm built upon it. The algorithm reduces the satisfiability problem of EUF^n formulas to the satisfiability of existential Presburger arithmetic with divisibility by leveraging the structure of the *CCG*. Since the satisfiability of such Presburger formulas is decidable [Lechner et al. 2015], the satisfiability of EUF^n formulas is also decidable.

3.1 Conditional Congruence Graph

Similar to the congruence closure algorithm for determining the satisfiability of standard EUF, the decision procedure for EUF^n also models terms in the formula as nodes and functional relationships between terms as directed edges. The primary distinction between the two approaches lies in the conditional labels attached to the edges of the *CCG*. These labels specify the conditions under which functional or equivalence relations hold. Consequently, when handling equivalence relations, the algorithm maintains conditional equivalence edges rather than merging equivalent nodes.

For simplicity, our analysis in this section is limited to formulas involving unary functions. Nevertheless, the proposed approach can be readily extended to handle multi-arity functions with minimal modification. Note that while EUF^n does not support parametric composition of multi-arity functions, constant-depth compositions of such functions, as allowed in standard EUF syntax, are still supported in EUF^n. We begin by formally defining the *CCG*.

Definition 2 (Conditional Congruence Graph). Let $\mathbb{C}$ denote the set of conditions (logical formulas), $\mathbb{E}$ the set of linear expressions, and $\mathbb{F}$ the set of unary function symbols (including finite compositions).

A **conditional congruence graph (CCG)** is a triple $(V, E_{\equiv}, E_{\succ})$, where:

- V is the set of nodes representing terms.
- $E_{\equiv} \subseteq V \times \mathbb{C} \times V$ is the set of **conditional equivalence edges**. An edge $(u, \phi, v) \in E_{\equiv}$ (undirected) denotes that u and v are equivalent under condition $\phi \in \mathbb{C}$.
- $E_{\succ} \subseteq V \times (\mathbb{C} \times \mathbb{F} \times \mathbb{E}) \times V$ is the set of **conditional superterm edges**. An edge $(v, (\phi, f, k), u) \in E_{\succ}$ (directed) denotes that $v = f^k(u)$ under condition $\phi \in \mathbb{C}$, where $f \in \mathbb{F}$ represents a functional relationship between terms and $k \in \mathbb{E}$ specifies the composition depth of the function.

For any term $t \in V$, we define $[t]$ as the set containing t itself and all nodes conditionally equivalent to t; that is, for every $t' \in [t] \setminus \{t\}$, there exists an edge $(t, \phi, t') \in E_{\equiv}$. Note that when the directed edges and conditions on equivalence edges are ignored, the subgraph induced by the nodes in $[t]$ forms a complete graph.

Definition 3 (f-*term*). Let t be an arbitrary term and let $f \in \mathbb{F}$ be a unary function. The f-*term* of t, denoted $f\text{-}term(t)$, is the set of all terms obtained by iteratively applying f to t (for any number of applications). Formally,

$$f\text{-}term(t) = \bigcup_{k \in \mathbb{E}} \{f^k(t)\}.$$

For brevity, we use f-t to denote $f\text{-}term(t)$ in subsequent discussions.

3.2 Overview of the Algorithm

As discussed earlier, the *CCG* captures both conditional superterm relations and conditional equivalence relations between terms. Therefore, the algorithm for solving EUF^n formulas based on the *CCG* first processes all equalities in the formula to derive conditional equivalence relations among terms, maintaining conditional consistency after each equivalence relation update. It then evaluates the disequalities to verify whether there exists a non-conflicting assignment. If such an assignment exists, instantiating the original EUF^n formula under this assignment yields a satisfiable EUF instance; otherwise, the EUF^n formula is unsatisfiable.

When processing equations, the proposed algorithm does not immediately merge equivalent nodes, but instead updates the conditional equivalence and superterm relations between them. As illustrated in Section 2.3, processing each equality in an EUF^n formula involves the following three

steps: (1) **Equivalence propagation**: Enforce the transitivity of equality, similar to the *Union* operation in standard EUF solving algorithms. (2) **Superterm inheritance**: After updating equivalence relations, propagate the corresponding superterm (and subterm) relationships. (3) **Conditional congruence closure**: Update the conditional equivalence relations between congruence terms based on the congruence axiom and generalized congruence rules (Theorem 1). As illustrated in Section 2.3, the above procedure terminates because no term can be its own superterm, and is correct because the *CCG* completely and accurately captures the congruence graphs of the infinitely many EUF formulas corresponding to the EUF^n formula.

Given that an EUF^n formula adheres to the syntax in Figure 2, it can be transformed into disjunctive normal form (DNF), possibly incurring an exponential size increase [Kroening and Strichman 2008]. The validity of the formula is then determined by the satisfiability of its DNF conjuncts, where each conjunct consists of EUF^n-literals (atomic formulas or their negations). For each conjunct, Algorithm 1 either produces a satisfying assignment for the natural number variables or concludes UNSAT.

Algorithm 1 outlines the process for resolving the target conjunct. In Line 1, the algorithm initializes a *CCG*; for example, the graph shown in Figure 3a is the resulting initial *CCG*. The detailed functionality of the initialization procedure, `Initialize`, will be elaborated in Section 3.3. In Line 2, the queue W is initialized to track conditional equivalence edges on which conditional congruence closure has not yet been applied. In Lines 3-9, the algorithm processes equalities in Ψ (*e.g.*, $term_l$=$term_r$). Lines 4 perform **equivalence propagation** by invoking the `EqProp` function. This function first executes Line 17, which calls `AddEq` to insert a conditional equivalence edge between $term_l$ and $term_r$. As illustrated in the example in Section 2.3, an equivalence edge is added between x and y in Figure 3b.

Next, Lines 19–20 iteratively invoke `AddEq` for $term_1$ and $term_2$ where $term_1$ and $term_2$ are connected to $term_l$ and $term_r$ via conditional equivalence edges, respectively, *i.e.*, $term_1 \in [term_l] \setminus \{term_l\}$ and $term_2 \in [term_r] \setminus \{term_r\}$. These calls update equivalence edges between $term_1$ and $term_2$, establishing their equivalence under condition $\phi_{\text{new}} \wedge \phi_1 \wedge \phi_2$. Note that during loop execution, a pre-loop snapshot of $E_{\equiv}$ is saved and used exclusively throughout all iterations, thereby ensuring termination. In the example from Section 2.3, since $[x] = \{x\}$ and $[y] = \{y\}$, no new equivalence edges are added at this step. The `AddEq` function (Lines 21–24 in Algorithm 1) incorporates a new conditional equivalence edge $(term_l, \phi_{\text{new}}, term_r)$ into the *CCG*. It first checks whether a conditional equivalence edge already exists between $term_l$ and $term_r$ (Line 22); if not, ϕ_{old} is set to **false**. Line 23 updates the equivalence condition to $\phi_{\text{old}} \vee \phi_{\text{new}}$. Note that after adding a conditional equivalence edge, the `AddEq` function does not recursively call itself to propagate the equivalence further. This is because $[term_1] = [term_l]$ and $[term_2] = [term_r]$ (Line 19), so no additional iterations are required to establish equivalence between $[term_1]$ and $[term_2]$, thus ensuring termination.

Since the equivalence condition has changed, Line 24 invokes the `InheritST` function to perform **superterm inheritance**, updating the superterms/subterms of $term_l$ and $term_r$ along with their associated conditions. That is, for every $(term_1, (\phi_1, f_1, k_1), term_l) \in E_{\succ}$ and $(term_l, (\phi_2, f_2, k_2), term_2) \in E_{\succ}$, add or update $(term_1, (\phi_1 \wedge \phi_{\text{new}}, f_1, k_1), term_r)$ and $(term_r, (\phi_2 \wedge \phi_{\text{new}}, f_2, k_2), term_2)$ to $E_{\succ}$. Moreover, for any other superterm $term_s$ of $term_r$ with function f_1, the superterm edges between $term_1$ and $term_s$ are updated accordingly; similarly, for any other superterm $term_s$ of $term_2$ with function f_2, the superterm edges between $term_r$ and $term_s$ are updated accordingly. Finally, the superterms and subterms of $term_r$ are similarly propagated to $term_l$. In the example from Section 2.3, Figure 3c shows the resulting graph after executing `InheritST`(x, true, y). It can be seen that the superterm z of x also becomes a superterm of y, and the conditional superterm relation between z and y's original superterm k is also added. Similarly, y's superterm k is inherited by x in the same manner.

Algorithm 1: Solve

Input: A finite conjunction Ψ of EUF^n-literals.
Output: A solution or UNSAT.

1 Let $CCG\ (E, E_{\equiv}, E_{\succ}) \leftarrow$ `Initialize`(Ψ) `// Section 3.3`
2 Let $W \leftarrow \emptyset$ `// A global stack-based worklist`
3 **foreach** $term_l = term_r \in \Psi$ **do**
  `// Equivalence propagation`
4 `EqProp`$(term_l, \textbf{true}, term_r)$
5 **while** $W \neq \emptyset$ **do**
6 $(term'_l, \phi', term'_r) \leftarrow W.pop()$
  `// Conditional congruence closure`
7 $\widehat{E}_{\equiv} \leftarrow$ `CongruenceClosure`$(term'_l, \phi', term'_r)$ `// Section 3.5`
8 **foreach** $(term'_1, \phi'', term'_2) \in \widehat{E}_{\equiv}$ **do**
9 `EqProp`$(term'_1, \phi'', term'_2)$
10 $\Phi \leftarrow \textbf{true}$
11 **foreach** $\neg(term_l = term_r) \in \Psi$ **do**
12 **if** $(term_l, \phi, term_r) \in E_{\equiv}$ **then**
13 $\Phi \leftarrow \Phi \wedge (\neg\phi)$
14 *result* $\leftarrow$ `SolvePresburgerWithD`(Φ)
15 **return** *result*
16 **Function** `EqProp`$(term_l, \phi_{\text{new}}, term_r)$**:**
17 `AddEq`$(term_l, \phi_{\text{new}}, term_r)$
18 $W.push((term_l, \phi_{\text{new}}, term_r))$
  `//` $term_1 \in [term_l] \setminus \{term_l\}, term_2 \in [term_r] \setminus \{term_r\}$
19 **foreach** $(term_l, \phi_1, term_1) \in E_{\equiv}, (term_r, \phi_2, term_2) \in E_{\equiv}$ **do**
20 `AddEq`$(term_1,\ \phi_{\text{new}} \wedge \phi_1 \wedge \phi_2,\ term_2)$
21 **Function** `AddEq`$(term_l, \phi_{\text{new}}, term_r)$**:**
22 **if** $(term_l, \phi_{old}, term_r) \in E_{\equiv}$ **then**
23 $E_{\equiv} \leftarrow E_{\equiv} \setminus \{(term_l, \phi_{\text{old}}, term_r)\} \cup \{(term_l, \phi_{\text{old}} \vee \phi_{\text{new}}, term_r)\}$
  `// Superterm inheritance`
24 `InheritST`$(term_l, \phi_{\text{new}}, term_r)$

In the while loop from Lines 5–9, Line 6 pops an unprocessed edge from W. Lines 7–9 then perform the **conditional congruence closure**, deriving new equivalences between congruent terms via the `CongruenceClosure` procedure, whose implementation is described in Section 3.5. The resulting conditional equivalences are collected in the edge set $\widehat{E}_{\equiv}$ (Line 7) and propagated via the `EqProp` function. In Section 2.3, after processing $x = y$, we have $\widehat{E}_{\equiv} = \{(z, m = n \wedge m > 0 \wedge n > 0, k)\}$. As a result, a conditional equivalence edge between z and k is added via `EqProp`, as illustrated in Figure 3d. Furthermore, the triple $(z, m = n \wedge m > 0 \wedge n > 0, k)$ is pushed into W. Note that the equivalence relations added to the *CCG* via `AddEq` (Lines 19–20), which are of the form $(term_1, \phi_{\text{new}} \wedge \phi_1 \wedge \phi_2, term_2)$ where $term_1 \in [term_l] \setminus \{term_l\}$ and $term_2 \in [term_r] \setminus \{term_r\}$, are not considered for further congruence closure. This is because $term_1$ (respectively, $term_2$) shares

identical superterms with $term_l$ (respectively, $term_r$). Consequently, for the equivalent condition $\phi \vee \phi_{\text{new}} \vee (\phi_{\text{new}} \wedge \phi_1 \wedge \phi_2)$ between congruence terms obtained by considering both edges, the condition introduced by the edge $(term_1, \phi_{\text{new}} \wedge \phi_1 \wedge \phi_2, term_2)$ is redundant. (For brevity, we omit the superterm conditions since the conclusion holds when they are considered.)

The algorithm then processes the disequalities in Ψ (Lines 11–13). For each disequality $term_l \neq term_r$ (Line 11), the algorithm identifies from the *CCG* the condition ϕ under which $term_l$ and $term_r$ become equivalent (Line 12). Since a conflict arises if ϕ holds, its negation $\neg\phi$ is conjoined to the global condition Φ (Line 13). After processing all disequalities, Φ encapsulates the constraints necessary to prevent conflicts. Thus, if Φ is satisfiable, the original EUF^n formula is satisfiable; otherwise, no satisfying assignment exists (Line 14). In the example from Section 2.3, when processing $z \neq k$, the equivalence condition between z and k is found to be $\phi = (m = n \wedge m > 0 \wedge n > 0) \vee (m = 0 \wedge n = 0)$ (Figure 3d, which omits some conditions for brevity as noted earlier). Since $\neg\phi$ is satisfiable, the EUF^n formula in Section 2.3 is satisfiable. Note that Φ can be expressed as an existential sentence in Presburger arithmetic with divisibility, a theory with decidable satisfiability. A formal justification for this characterization is provided in Section 3.5 and Section 3.6.

3.3 Initialization

This subsection details the initialization function `Initialize` (Line 1) in Algorithm 1. During initialization, all terms from the EUF^n formula Ψ are collected into set V, which serves as nodes of the *CCG*. The superterm edge set $E_{\succ}$ is initialized by defining relations where one term can be obtained by applying functions to another (*e.g.*, $f^m(x)$ becomes a superterm of $f^n(x)$ via $(m\text{-}n)$ applications of f, under condition $m > n$). The sets V and $E_{\succ}$ are initialized as follows:

$$\frac{term \in \Psi}{V \leftarrow V \cup \{term\}} \qquad \frac{f^{k_1}(term), f^{k_2}(term), term \in V, \quad f \in \mathbb{F}}{E_{\succ} \leftarrow E_{\succ} \cup \{(f^{k_2}(term), (k_2 > k_1, f, k_2 - k_1), f^{k_1}(term))\}}$$

This rule states that if the conditions above the line hold, then the operations below the line are executed. Since the syntax of EUF^n permits function composition ($f = f_1 \circ \cdots \circ f_n$; Section 2.1), each addition of a superterm edge in the *CCG* (*e.g.*, for function relation f_i) triggers an immediate check. This check determines if new superterm edges (*e.g.*, for f) can be added based on function composition rules and the edge's predecessor/successor relationships, with corresponding superterm conditions and composition counts derived from intermediate edges. For example, if $x = f_1(y)$, $y = f_2(z)$, and $f = f_1 \circ f_2$, then a superterm edge representing $x = f(z)$ is added. This composition check is intrinsic to all superterm edge additions in subsequent algorithm steps.

Finally, the conditional equivalence edge set $E_{\equiv}$ is initialized with conditional equivalence relations between terms that are derivable from identical base terms through identical function applications of matching depth (*i.e.*, according to Theorem 1 and the congruence axiom (4)). The initialization rules are as follows:

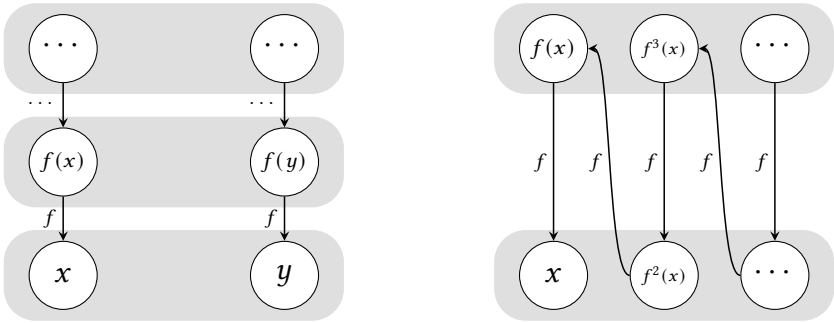
$$\frac{\begin{array}{ll}(term_1, (k_1 > 0, f, k_1), term) \in E_{\succ}, & term, term_1 \in V, \\ (term_2, (k_2 > 0, f, k_2), term) \in E_{\succ}, & term_2 \in V, f \in \mathbb{F}\end{array}}{E_{\equiv} \leftarrow E_{\equiv} \cup \{(term_1, k_1 = k_2, term_2)\}}$$

After initializing the equivalence edges in $E_{\equiv}$, no further congruence closure processing is required. This is because the superterms of $term_1$ and $term_2$ are also superterms of $term$, and their conditional equivalence relations are already included in $E_{\equiv}$.

3.4 Periodic Equivalence

Before introducing the function `CongruenceClosure` (Line 7) that handles congruence relations in Algorithm 1, we first introduce a phenomenon we refer to as *periodic equivalence*. In the standard

congruence closure algorithm for EUF formulas, after performing the $Union(x, y)$ operation, the algorithm recursively merges congruent terms by proceeding upward and executing $Union(f(x), f(y))$, $Union(f(f(x)), f(f(y)))$, and so on. The resulting equivalence classes can be categorized into two types based on whether a superterm relationship exists between x and y, as illustrated in Figure 4, where shaded nodes belong to the same equivalence class. Figure 4a depicts the case where no superterm relationship exists between x and y; that is, the upward search for congruent terms starting from x does not reach y (and vice versa). We refer to the equivalence classes formed in this case as having a **hierarchical structure**. Figure 4b illustrates the case where a superterm relationship exists between x and y, such as in $Union(x, f^2(x))$. Here, the upward search from x eventually reaches $f^2(x)$, causing equivalence classes from different levels to merge, forming what we call a **partitioned structure**.



(a) Without superterm: Hierarchical structure (b) With superterm: Partitioned structure

Fig. 4. Impact of superterm relationships on equivalence class formation.

In EUF, where the depth of function applications is bounded, both cases in Figure 4 can be handled uniformly by recursively merging until a fixed point is reached. However, in EUF^n, where the composition depth of functions may be represented by variables, this recursive approach fails. Thus, we must treat the two cases separately. For instance, in the hierarchical structure (Figure 4a), we have $x = y \wedge m = n \rightarrow f^n(x) = f^m(y)$, while in the partitioned structure (Figure 4b), we have $m \equiv_2 n \rightarrow f^n(x) = f^m(x)$ (for simplicity, we use $m \equiv_2 n$ to denote $m \equiv n \mod 2$, and adopt this notation throughout). We refer to the equivalence relation arising from a superterm relationship between x and y as *periodic equivalence*. This section elaborates on the treatment of such congruence relations in solving EUF^n formulas.

Definition 4 (Periodic Equivalence). Given an equivalence pair $(term_l, term_r)$ where $term_l$ and $term_r$ are distinct terms, if

$$\exists f \in \mathbb{F}.\ term_l \in f\text{-}term_r \vee term_r \in f\text{-}term_l$$

holds, then $(term_l, term_r)$ is called a periodic equivalence pair and $term_l$ and $term_r$ are considered periodically equivalent.

Example 1. Suppose the equation to be processed is $x = f^2(x)$. By definition, x and $f^2(x)$ are periodically equivalent, as shown in Figure 4b, this equivalence relation partitions the set $f\text{-}x$ into two equivalence classes: $\{x, f^2(x), \cdots\}$ and $\{f(x), f^3(x), \cdots\}$, each containing infinitely many elements. Therefore, after adding the equivalence edge $(x, \text{True}, f^2(x))$ to the *CCG*, for any two elements $f^n(x) \in f\text{-}x$ and $f^m(x) \in f\text{-}x$ that belong to the same equivalence class, an equivalence edge $(f^n(x), m \equiv_2 n, f^m(x))$ should be added.

Example 2. Assume we are processing the equation $f^6(x) = f^8(x)$, having previously processed the equation $x = f^5(x)$. Consequently, an equivalence edge with condition $6 \equiv_5 8$ already exists between nodes $f^6(x)$ and $f^8(x)$ in the *CCG*. Since $\neg(6 \equiv_5 8)$ holds, the equivalence class derived from the relation $(x, f^5(x))$ is distinct from that derived from $(f^6(x), f^8(x))$. Given

$$\begin{pmatrix} x = f^5(x) \Rightarrow x = f^5(x) = f^{10}(x) \\ f^6(x) = f^8(x) \Rightarrow f^6(x) = f^8(x) = f^{10}(x) \end{pmatrix} \Rightarrow x = f^5(x) = f^6(x) \Rightarrow x = f(x)$$

where the first implication follows from applying f^5 to $x = f^5(x)$, yielding $f^5(x) = f^{10}(x)$, the second follows similarly, and the last from $f(x) = f(f^5(x))$. We therefore conclude that the equivalence classes derived from the relations $(x, f^5(x))$ and $(f^6(x), f^8(x))$ are identical to those derived from $(x, f(x))$. A more general and formal treatment will be provided in Section 3.5.

3.5 Congruence Relations Processing

In this subsection, we explain how to handle congruence relations after adding an equivalence edge, *i.e.*, the `CongruenceClosure` function (Line 7) in Algorithm 1. Suppose the added equivalence edge is $(\boldsymbol{term_l}, \boldsymbol{\phi}, \boldsymbol{term_r})$. We will discuss the cases based on whether a *periodic equivalence* exists between $term_l$ and $term_r$. During this process, we use $\widehat{E}_{\equiv}$ to record the conditional equivalence relations derived from handling the congruence relations. The computation rules presented later in this subsection state that if the conditions above the line hold, then the operations below the line are executed.

$term_l$ and $term_r$ are not periodically equivalent. We first consider the case where the newly added equivalence relation is not a *periodic equivalence*, meaning there is no superterm relation between $term_l$ and $term_r$. Since $term_l$ and $term_r$ are equivalent, according to the congruence axiom (4) and Theorem 1, for any $f \in \mathbb{F}$, we need to identify terms within $f\text{-}term_l$ and $f\text{-}term_r$ that can be derived as equivalent. The specific rule for adding equivalence relations is as follows. The condition $\phi_1 \wedge \phi_2$ ensures that both superterm edges can hold simultaneously, prevents an infinite upward search for congruent terms, and thereby guarantees algorithm termination. The validity of the equivalence condition between terms is ultimately checked during the final satisfiability check of the formula.

$$\frac{\begin{array}{c}(term_l, \phi, term_r) \in E_{\equiv}, \quad (term_1, (\phi_1, f, k_1), term_l) \in E_{\succ}, \\ (term_2, (\phi_2, f, k_2), term_r) \in E_{\succ}, \quad (term_1, \phi_{\text{old}}, term_2) \in E_{\equiv}, \quad \widehat{\phi} = \phi_1 \wedge \phi_2 \text{ is satisfiable.}\end{array}}{\widehat{E}_{\equiv} \leftarrow \widehat{E}_{\equiv} \cup \{(term_1, (\phi \wedge \widehat{\phi} \wedge k_1 = k_2) \vee \phi_{\text{old}}, term_2)\}} \tag{3}$$

$term_l$ and $term_r$ are periodically equivalent. When a superterm relationship exists between $term_l$ and $term_r$ (assuming $term_r = f^k(term_l)$ for $k > 0$), the addition of an equivalence relation partitions the set $f\text{-}term_l$ into k equivalence classes. First, for elements outside the set $f\text{-}term_l$, we handle them according to Rule (3); that is, for any $g \in \mathbb{F}$ with $g \neq f$, the elements in $\widehat{E}_{\equiv}$ follow Rule (3) with the function symbol f replaced by g. Next, we discuss the elements in the set $f\text{-}term_l$, as discussed in Examples 1 and 2, we will consider two cases: whether a *periodic equivalence* involving $term_l$ and $term_r$ has already been processed.

(1) **$(term_l, term_r)$ is the first added *periodic equivalence* pair with respect to $f\text{-}term_l$.** For elements in $f\text{-}term_l$, any two elements that belong to the same equivalence class are

considered equivalent and processed according to the following rules:

$$\frac{\begin{array}{c}(term_l, \phi, term_r) \in E_{\equiv}, \quad (term_r, (\phi_1, f, k), term_l) \in E_{\succ}, \\ (term_1, (\phi_2, f, k_1), term_l) \in E_{\succ}, \quad (term_2, (\phi_3, f, k_2), term_l) \in E_{\succ}, \\ (term_1, \phi_{\text{old}}, term_2) \in E_{\equiv}, \quad \widehat{\phi} = \phi_1 \wedge \phi_2 \wedge \phi_3 \text{ is satisfiable.}\end{array}}{\widehat{E}_{\equiv} \leftarrow \widehat{E}_{\equiv} \cup \{(term_1, (\phi \wedge \widehat{\phi} \wedge k_1 \equiv_k k_2) \vee \phi_{\text{old}}, term_2)\}} \tag{4}$$

Note that, modular arithmetic can be expressed with an existential sentence of Presburger arithmetic with divisibility, as shown below:

$$m \equiv_k n \Leftrightarrow \exists c.\, k \mid (m - c) \wedge k \mid (n - c) \wedge c < k$$
$$\neg(m \equiv_k n) \Leftrightarrow \exists c.\, k \mid (m - c) \wedge \neg(k \mid (n - c)) \wedge c < k$$

(2) **The *periodic equivalence* relation $(\text{term}_{l'}, \text{term}_{r'})$ with respect to f-$term_l$ has been processed.** In this case, an equivalence edge already exists between $term_l$ and $term_r$. The corresponding *periodic equivalence* pair $(term_{l'}, term_{r'})$ responsible for introducing this edge can be directly retrieved from the algorithm's execution trace. Consequently, the newly introduced *periodic equivalence* $(term_l, term_r)$ will further refine the equivalence classes partitioned by $(term_{l'}, term_{r'})$. In Theorem 4, we prove that multiple *periodic equivalences* can be represented by a single *periodic equivalence*, and both yield identical equivalence class partitions. Without loss of generality, we assume $term_{l'} = f^{k_1}(term_l)$ and $term_{r'} = f^{k_2}(term_l)$, with $k_2 > k_1$. The specific processing rules are as follows, where gcd denotes the greatest common divisor function:

$$\frac{\begin{array}{c}(term_l, \phi, term_r) \in E_{\equiv}, \quad (term_r, (\phi_1, f, k), term_l) \in E_{\succ}, \\ (term_{l'}, (\phi_2, f, k_1), term_l) \in E_{\succ}, \quad (term_{r'}, (\phi_3, f, k_2), term_l) \in E_{\succ}, \\ (term_{r'}, (\phi_4, f, k_2 - k_1), term_{l'}) \in E_{\succ}, \quad (term_1, (\phi_5, f, k_3), term_l) \in E_{\succ}, \\ (term_2, (\phi_6, f, k_4), term_l) \in E_{\succ}, \quad (term_1, \phi_{\text{old}}, term_2) \in E_{\equiv}, \\ \widehat{\phi} = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5 \wedge \phi_6 \text{ is satisfiable.}\end{array}}{\widehat{E}_{\equiv} \leftarrow \widehat{E}_{\equiv} \cup \{(term_1, (\phi \wedge \widehat{\phi} \wedge k_3 \equiv_{\gcd(k,\ (k_2 - k_1))} k_4) \vee \phi_{\text{old}}, term_2)\}} \tag{5}$$

The correctness of this rule is established by Theorem 4 in Section 3.6. Noted that, according to Bézout's identity [Rosen and Krithivasan 1999], the gcd function can be expressed using an existential sentence of Presburger arithmetic with divisibility, as shown below:

$$a = \gcd(m, n) \Leftrightarrow \exists b.\, a \mid m \wedge a \mid n \wedge m \mid b \wedge n \mid (b - a)$$

Above, we described the handling of `CongruenceClosure()`, which is invoked after each addition of an equivalence relation through the `EqProp` function. The resulting conditional equivalences are stored in $\widehat{E}_{\equiv}$ and subsequently added to the *CCG*.

Theorem 2 (Decidability). For any finite conjunctive formula Ψ of EUF^n equalities and disequalities, its satisfiability is decidable via the *CCG* algorithm.

Proof. For an arbitrary formula Ψ, the *CCG* algorithm transforms it into Φ such that Ψ is satisfiable if and only if Φ is satisfiable. The formula Φ is an existential sentence of Presburger arithmetic with divisibility. This is because Algorithm 1 introduces existential quantifiers only through the modulo and gcd operations. Although Line 13 involves negation, it introduces no universal quantifiers, as both $m \equiv_a n$ and its negation are expressible as existential sentences. The gcd function, having no negation semantics, similarly avoids universal quantification. Therefore, the formula Φ produced by Algorithm 1 is an existential sentence of Presburger arithmetic with

divisibility. Since this logic is decidable [Lechner et al. 2015], the satisfiability of Ψ is also decidable. □

3.6 Correctness and Complexity

To prove the correctness of Algorithm 1, we first demonstrate its termination.

Theorem 3 (Termination). Algorithm 1 terminates on any finite input conjunct Ψ, where Ψ is a conjunction of EUF^n equalities and disequalities.

Proof. Since the size of the Ψ is finite, it suffices to consider whether the processing of each individual formula terminates. Lines 11–13 handle disequalities by collecting the associated equivalence conditions of the involved terms, which is clearly a terminating process.

For equalities, the processing involves three steps. The first step, **equivalence propagation** (Line 4), updates the conditional equivalence relations between elements in $[term_l]$ and $[term_r]$. As the number of nodes in $[term_l]$ and $[term_r]$ is finite, this step terminates. Following the insertion of this equivalence edge, the second step, **superterm inheritance**, is performed. This step terminates because the number of superterms and subterms of $term_l$ and $term_r$ is finite. The third step, **conditional congruence closure**, derives new equivalences based on the congruence axiom (4) and the generalized congruence theorem (Theorem 1), which may in turn trigger additional congruence-based deductions. However, since the number of nodes is finite and no term can be a superterm of itself (as such conditions are unsatisfiable), no infinite superterm chains can arise. Thus, this process is guaranteed to terminate. Therefore, we conclude that Algorithm 1 terminates. □

We now prove the soundness and completeness of Algorithm 1, namely, that the input EUF^n formula Ψ is unsatisfiable if and only if the existential Presburger formula Φ produced by the algorithm is unsatisfiable. To establish this, we first prove Lemma 1, as stated below:

Lemma 1. After Algorithm 1 completes, for every equivalence edge $(term_l, \phi, term_r)$ in the *CCG* and any assignment $\sigma : \mathsf{N} \to \mathbb{N}$, the following invariant holds: ϕ is satisfiable under σ if and only if $term_l$ and $term_r$ belong to the same equivalence class in the congruence closure of $\Psi[\sigma]$.

Proof. We prove by induction that after processing each equation, the *CCG* and the congruence closure of $\Psi[\sigma]$ obtained after this processing satisfy the invariant stated in Lemma 1. The detailed proof can be found in Appendix B; below we outline the main ideas.

First, during Algorithm 1's initialization (Section 3.3), equivalence edges are generated solely from terms obtained by applying the same function identically to identical terms (congruence axiom (4) and Theorem 1). The initialization rules (*i.e.*, the condition $k_1 = k_2$) define the initial congruence closure for all $\Psi[\sigma]$, hence the invariant stated in Lemma 1 holds at initialization.

Assume the invariant of Lemma 1 holds after the first k equations. For the $(k{+}1)$-th equation, the algorithm adds equivalence edges via `EqProp`, which first adds an edge based on its parameters (Lines 4 and 9), then adds edges for transitivity (Line 20). Parameters come from two sources: the $(k{+}1)$-th equation itself, and edges from `CongruenceClosure` (Line 9). Edges from the equation satisfy the invariant; those from `CongruenceClosure` use Rules (3)–(5), whose correctness is shown later. The proof for transitivity edges (added by `AddEq`) is given in Appendix B. □

The `CongruenceClosure` function derives new equivalence edges using Rules (3)–(5). While Rules (3)–(4) trivially satisfy the invariant in Lemma 1 (*e.g.*, Rule (3) directly encodes the congruence axiom in solving $\Psi[\sigma]$), we now prove that Rule (5) also preserves this invariant.

Lemma 2. Assume $n > m$, $term = f^m(term) = f^n(term) \leftrightarrow term = f^{\gcd(m,n)}(term)$.

Proof. The detailed proof can be found in Appendix C. □

Theorem 4. Given two periodic equivalence relations $term = f^k(term)$ and $f^m(term) = f^n(term)$. Then

$$term = f^k(term) \wedge f^m(term) = f^n(term) \leftrightarrow term = f^{\gcd(k,(n-m))}(term).$$

Proof. The detailed proof can be found in Appendix C. □

Assuming that all current *CCG* edges satisfy the invariant in Lemma 1, Theorem 4 ensures that Rule (5) precisely characterizes the equivalence class partitioning induced by multiple periodic equivalences. Therefore, the addition of Rule (5) also preserves the invariant in Lemma 1.

Theorem 5 (Soundness and Completeness of Algorithm 1). For any finite conjunctive formula Ψ of EUF^n equalities and disequalities, let Φ be the existential formula in Presburger arithmetic with divisibility generated by Algorithm 1. Then

$$\Psi \text{ is satisfiable} \iff \Phi \text{ is satisfiable.}$$

Lemma 1 establishes that the *CCG* precisely captures all equivalence relations in $\Psi[\sigma]$. This directly implies the correctness of Algorithm 1, as stated in Theorem 5.

We now analyze the complexity of Algorithm 1.

Theorem 6 (Complexity). Algorithm 1 decides the satisfiability of a conjunction Ψ of EUF^n equalities and disequalities of length n in 2NEXPTIME.

Proof. For an EUF^n formula Ψ and its instantiation yielding the disjunctive form of infinitely many EUF formulas Ψ' (*e.g.*, Equation (2)), each EUF^n term with a *parametric composition depth* corresponds to infinitely many EUF terms. However, the number of possible superterm orderings is bounded by $n!$. For example, three terms $f^{m_1}(x)$, $f^{m_2}(x)$, and $f^{m_3}(x)$ yield 3! possible orderings of m_1, m_2, and m_3, each inducing a distinct superterm chain. For each ordering, a chain of length n requires at most $(n-1)$ *Union* operations, each corresponding to one `EqProp` execution. Consequently, Algorithm 1 executes $O(n! \cdot n)$ `EqProp` operations, where each `EqProp` invocation calls `AddEq` $O(n^2)$ times, and each `AddEq` call updates $O(n^2)$ superterm relations at Line 24. The cumulative cost of these internal operations is $O(n! \cdot n^5)$, which is asymptotically dominated by the subsequent solving of the generated formula and thus does not affect the overall complexity bound.

Each ordering is associated with corresponding constraints (*e.g.*, $m_1 > m_2 > m_3$), which are incorporated into the equivalence conditions between terms. Upon termination of Algorithm 1, these conditions may form an existential Presburger formula with divisibility of length $O(n!)$. The satisfiability problem for such formulas is in NEXPTIME [Lechner et al. 2015], meaning that, on a nondeterministic Turing machine, a formula of length L can be decided in time $2^{p(L)}$ for some polynomial p. In our case, $L = O(n!)$, so the solving time becomes $2^{p(O(n!))} = 2^{O((n!)^c)}$ for some constant c. Since $n! = 2^{O(n \log n)}$, we have $(n!)^c = 2^{O(n \log n)}$. Thus, the total time is $2^{2^{O(n \log n)}}$, *i.e.*, $2^{2^{p(n)}}$ for some polynomial p. Hence, the complexity of solving the resultant formula is 2NEXPTIME (*i.e.*, solvable in $O(2^{2^{p(n)}})$ time on a nondeterministic Turing machine, for some polynomial p). Additionally, each call to `CongruenceClosure` (Line 7) must verify the satisfiability of superterm conditions. Due to superterm inheritance, these conditions incorporate term equivalence relations, potentially involving divisibility. However, checking only the satisfiability of the ordering relation along the superterm chain (*e.g.*, $m_1 > m_2 > \cdots$) suffices to ensure that no term becomes its own superterm, thereby guaranteeing termination. Since each such check requires solving an existential Presburger formula (NP-complete) and $O(n! \cdot n)$ checks are required, this step is in EXPTIME. Thus, the overall complexity of Algorithm 1 is 2NEXPTIME. □

4 Applications of EUF^n

This section demonstrates the applications of EUF^n through two case studies: the single-source single-target interleaved Dyck reachability problem and the verification of uninterpreted programs.

4.1 Single-Source Single-Target Interleaved Dyck Reachability

In this subsection, we formalize the encoding of a subclass of the single-source single-target interleaved Dyck reachability problem (IDRP) in EUF^n and prove that this subclass is decidable.

4.1.1 Background. We first introduce the concepts of IDRP and motivate our EUF^n encoding.
Regular Expression. A regular expression over an alphabet $\Sigma_{\mathcal{R}}$ is defined recursively as:

$$\mathcal{R} ::= \epsilon \mid a \mid (\mathcal{R}) \mid \mathcal{R}_1 \cdot \mathcal{R}_2 \mid \mathcal{R}_1 + \mathcal{R}_2 \mid \mathcal{R}^*$$

where ϵ denotes the empty string and $a \in \Sigma_{\mathcal{R}}$ represents a terminal symbol, $(\mathcal{R})$ denotes expression grouping to explicitly specify precedence or grouping, $\mathcal{R}_1 \cdot \mathcal{R}_2$ represents concatenation, $\mathcal{R}_1 + \mathcal{R}_2$ signifies alternation, $\mathcal{R}^*$ indicates zero or more repetitions. Operator precedence (descending): Kleene star (*), concatenation ($\cdot$), alternation (+).
Dyck Language. The Dyck language, a context-free language (CFL) class for bracket-matching, is formally defined by the production rule $S \to SS \mid (_i S)_i \mid \epsilon$ over alphabet $\Sigma = \{(_i,)_i \mid i = 1, \ldots, k\}$. Here $(_i$ are termed *opening labels* and $)_i$ *closing labels*. The Dyck reachability problem determines existence of a path between graph nodes where the concatenated edge labels form a valid Dyck string. This formalism models critical program behaviors including function call/return pairs [Späth et al. 2019; Sridharan et al. 2005] and field access sequences [Späth et al. 2019], establishing its broad utility in program analysis.
Single-Source Single-Target IDRP. Given Dyck languages D_α and D_β, defined over alphabets Σ_α and Σ_β, respectively, a string s is belongs to an interleaved Dyck language, denoted as $s \in D_\alpha \odot D_\beta$, if its projections on Σ_α and Σ_β belong to D_α and D_β respectively (*i.e.*, $s \downharpoonright \Sigma_\alpha \in D_\alpha \wedge s \downharpoonright \Sigma_\beta \in D_\beta$). Given a graph $G = (V, E)$ where each edge $e \in E$ carries a label $\lambda(e) \in \Sigma_\alpha \cup \Sigma_\beta$, two nodes $s, t \in V$ are $D_\alpha \odot D_\beta$*-reachable* iff there exists a path from s to t whose edge label concatenation forms a string $p \in D_\alpha \odot D_\beta$. For instance, in Figure 5, nodes a and d are connected by a path labeled $(_1(_2[\![_1]\!])_2)_1]\!]_1$, demonstrating $D_\alpha \odot D_\beta$-reachability, where $\Sigma_\alpha = \{(_1, (_2,)_1,)_2\}$ and $\Sigma_\beta = \{[\![_1,]\!]_1\}$.

Furthermore, reachability evaluated solely between two specified points is termed single-source single-target interleaved Dyck reachability. While the interleaved Dyck reachability problem is undecidable [Reps 2000], its importance in static analysis applications such as context-sensitive data-dependence analysis [Reps 2000] and context- and field-sensitive data-flow analysis [Späth et al. 2019] has driven substantial research into approximation techniques [Conrado et al. 2025; Li et al. 2023; Späth et al. 2019; Zhang and Su 2017]. However, existing methods universally neglect the structural properties of connecting paths between reachable nodes.
Path Expressions. Given a graph G, all possible paths between any two nodes can be represented using a regular expression [Tarjan 1981]. Such expressions can be computed in $O(|E| \cdot \alpha(|E|))$ time, where $|E|$ is the number of edges and α is the inverse Ackermann function. For example, in Figure 5, the path expression between nodes a and d is represented by the regular expression $(_1((_2[\![_1]\!])_2)^*)_1]\!]_1$.

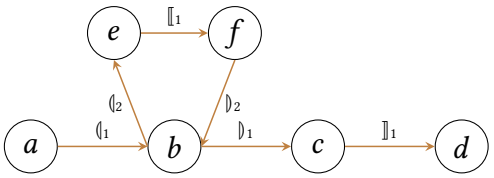


Fig. 5. An example of interleaved Dyck reachability.

Observation. We identify two structural patterns under which reachability is decidable: (1) path expressions contain no Kleene star operators; and (2) Kleene star operators are present, but they are neither:

- Nested within another Kleene star operator (*e.g.*, $((a)^*b)^*$), nor
- Combined with the alternation operator (*e.g.*, $(a + b)^*$)

The decidability of the single-source single-target IDRP is straightforward in case (1). For case (2), this paper establishes its decidability by encoding the problem using EUF^n. We further reveal that state-of-the-art approximation techniques—including SPDS [Späth et al. 2019], LCL [Zhang and Su 2017], ILP-based methods [Li et al. 2023], and d-MCFL(r) [Conrado et al. 2025]—fail to yield accurate results even under these constraints.

4.1.2 Decidability of the IDRP. We first outline the main idea of the encoding scheme. Nodes in the graph are encoded as variables, and labeled edges are encoded using uninterpreted functions. For example, a sequence such as $a \xrightarrow{(_1} b \xrightarrow{)_1} c$ is encoded as $a = f(b) \wedge c = f(b)$, where matching labels (*e.g.* $(_1/)_1$) are mapped to the same uninterpreted function. In this way, parenthesis matching is modeled as congruence reasoning, while the relative order of parentheses is captured by the composition order of uninterpreted functions. This idea closely resembles the encoding of bidirected Dyck reachability using EUF by Du *et al.* [Du et al. 2026b].

We restrict our scope to single-source single-target scenarios for two reasons. First, Interleaved Dyck Reachability is not an equivalence relation. In the example above, node a can reach node c, but not vice versa; yet the encoded formula makes variables a and c equivalent. Without this restriction, we cannot distinguish whether a reaches c or c reaches a. Second, branching in paths introduces another problem: edges such as $a \xrightarrow{(_1} b$ and $c \xrightarrow{)_1} b$ are directly encoded as $a = f(b) \wedge c = f(b)$, which incorrectly implies $a = c$ even though there is no reachability between them.

From the above encoding, a branch-free ("linear") unidirectional path can be encoded as an EUF formula. When the path contains loops (like Figure 5), we can introduce natural number variables (*e.g.*, n) to represent repetition counts, with different values corresponding to linear paths that traverse the loop different numbers of times. By assigning two variables (*e.g.*, v_{in} and v_{out}) representing the loop's entry and exit points, the path between v_{in} and v_{out} can be encoded using EUF^n formulas, where the extended expressive power of EUF^n enables the encoding of loops, as established in Theorem 7.

Note that the encoding is performed separately for D_α- and D_β-reachability between nodes. If the sets of reachable assignments for the loop count variables corresponding to D_α- and D_β-reachability intersect, then the combined assignment of all loop count variables corresponds to a path witnessing the D_α- and D_β reachability. We now illustrate the encoding with the $D_\alpha \odot D_\beta$-reachability problem between nodes a and d in Figure 5. The path expression between a and d is given by $\mathcal{R} = (_1((_2[\![_1)_2)^*)_1]\!]_1$, which contains a Kleene star structure. We introduce a variable n to represent the number of iterations of the loop. The final encoding using EUF^n is shown below, with D_α and D_β components separated for clarity:

$$\text{Encoding } D_\alpha : a = f_1(b_{\text{in}}) \wedge b_{\text{in}} = b_{\text{out}} \wedge c = f_1(b_{\text{out}}) \wedge c = d$$
$$\text{Encoding } D_\beta : a' = b'_{\text{in}} \wedge b'_{\text{in}} = g_1^n(b'_{\text{out}}) \wedge b'_{\text{out}} = c' \wedge g_1(c') = d'$$

The complete encoding procedure is detailed in the full version [Du et al. 2026a]. When solving the formula obtained from encoding D_α using the *CCG*, the condition for a to be equivalent to d is set to **true**. When solving the formula from encoding D_β, the condition for a' to be equivalent to d' is specified as $n = 1$. Since the conjunction **true** $\wedge\ n = 1$ is satisfiable, there exists a path such that nodes a and d are interleaved Dyck reachable, *i.e.*, a path that enters the loop exactly once.

Formally, Figure 6 defines the syntax of regular expressions under the above structural constraints, termed *WellFormed* regular expressions, where $a \in \Sigma_{\mathcal{R}}$. In this language, expressions prohibit nested Kleene stars and avoid alternation inside a Kleene star. If the path expression between nodes can be represented by a finite number of *WellFormed* expressions, *e.g.*, $\mathcal{R}_1 + \mathcal{R}_2$, we can encode $\mathcal{R}_1$ and $\mathcal{R}_2$ separately and take the disjunction of the resulting formulas. Having presented an intuitive encoding scheme for *WellFormed* regular expressions using EUF^n, we now prove that the corresponding IDRP is decidable when all path expressions between the target node pair are *WellFormed*.

$$\begin{aligned} \mathit{WellFormed} &::= \mathit{Term} \mid \mathit{Term} \cdot \mathit{Term} \\ \mathit{Term} &::= \mathit{Factor} \mid (\mathit{Factor})^* \\ \mathit{Factor} &::= \mathit{Atomic} \mid \mathit{Atomic} \cdot \mathit{Atomic} \\ \mathit{Atomic} &::= a \mid \epsilon \end{aligned}$$

Fig. 6. The Syntax of *WellFormed* regular expressions.

Theorem 7 (Decidability of the IDRP). Let $\mathcal{P}$ be the set of regular expressions encoding paths between two nodes u and v in a single-source single-target Interleaved Dyck reachability problem. If each expression $\mathcal{R} \in \mathcal{P}$ is *WellFormed*, then $\mathcal{P}$ can be encoded as a set of EUF^n formulas, thus establishing the decidability of the reachability problem.

Proof. We first demonstrate that any *WellFormed* regular expression $\mathcal{R}$ can be encoded as an EUF^n formula. As shown in the example in Figure 5, the primary challenge is automatically encoding Kleene star structures. We subsequently show how EUF^n formulas can express such structures, with the complete encoding algorithm for $\mathcal{R}$ provided in the full version [Du et al. 2026a]

Given a Kleene star expression $(s)^*$ and a natural number variable n, we remove the Dyck-matching pairs from s to obtain a simplified string s'. It can be shown that the encodings for $(s)^*$ and $(s')^*$ are equivalent. It is straightforward to detect whether s' contains any invalid pattern. The possible valid forms of s' fall into three categories: (1) a sequence of only *opening labels*, (2) a sequence of only *closing labels*, or (3) a sequence of *closing labels* followed by *opening labels*. Cases (1) and (2) can be encoded easily, and the final formula takes the form:

$$v_{in}/v_{out} = (f_1 \circ \cdots \circ f_k)^n (v_{out}/v_{in}),$$

where k is the length of s'.

Suppose in case (3), the string contains k_1 *closing labels* followed by k_2 *opening labels*. When $n = 1$, the functional relationship between v_{in} and v_{out} is easy to compute. When $n > 1$, assume without loss of generality that $k_1 > k_2$ (the case $k_1 \leq k_2$ follows analogously). If the k_2 *opening labels* do not match the k_2-length prefix of the k_1 *closing labels*, then only one valid path exists for $n = 1$. Otherwise, repeating s' n times yields a path that begins with k_1 *closing labels*, followed by $(k_1 - k_2)$ repeated *closing labels* for each additional iteration, and ends with k_2 *opening labels*. After introducing an intermediate variable v_{temp}, the corresponding functional relationship among v_{temp}, v_{in}, and v_{out} can be derived accordingly.

After encoding, the resulting EUF^n formula contains no periodic equivalences and consists solely of a conjunction of equalities. Solving this formula using the *CCG*-based method yields the equivalence conditions between the variables corresponding to the source and target nodes. These conditions form a quantifier-free Presburger formula, and its satisfiability problem is NP-complete. If this formula is satisfiable, then D_α- or D_β-reachability holds between the nodes. If the intersection of their solution sets is nonempty, then $D_\alpha \odot D_\beta$-reachability is established. □

However, existing methods—such as d-MCFL(r)-based reachability [Conrado et al. 2025], SPDS [Späth et al. 2019], LCL [Zhang and Su 2017], and ILP-based approaches [Li et al. 2023]—may still produce false positives even under the constraints of *WellFormed* regular expressions. First, consider the under-approximation technique based on d-MCFL(r) reachability [Conrado et al. 2025],

where parameters d and r control approximation precision (with larger values yielding tighter approximations for IDRP). While this strategy eliminates false positives, it inevitably introduces false negatives: for any fixed d and r, there exist Interleaved Dyck strings unrecognizable by d-MCFL(r), rendering it incapable of resolving certain IDRP instances even in loop-free cases. We now present counterexamples for the over-approximation algorithm under the same constraints.

SPDS [Späth et al. 2019] The synchronized pushdown system (SPDS) approximates IDRP by independently computing the reachability of the two Dyck languages involved in the interleaving. For example, given $L = D_\alpha \odot D_\beta$, SPDS considers s and t to be L-reachable if there are paths from s to t that are both D_α-reachable and D_β-reachable. However, this approximation may lead to false positives if the D_α-reachable and D_β-reachable paths are not the same, and there is no path that is both D_α-reachable and D_β-reachable, *i.e.*, no path that is interleaved Dyck reachable. For instance, suppose the path labels between nodes s and t form the following regular expression with only one layer of looping is

$$(\!|_1 (\!)_1 [\![_1)^*]\!]_1,$$

where $\Sigma_\alpha = \{(\!|_1,)\!)_1\}$ and $\Sigma_\beta = \{[\![_1,]\!]_1\}$. Since there exists a path $(\!|_1)\!)_1[\![_1]\!]_1 \downharpoonleft \Sigma_\alpha \in D_\alpha$ and a path $(\!|_1[\![_1]\!]_1 \downharpoonleft \Sigma_\beta \in D_\beta$, SPDS incorrectly concludes that s and t are interleaved Dyck reachable. However, upon further analysis, it becomes clear that only the path entering the loop once is D_α-reachable, and this path is not D_β-reachable. Thus, no interleaved Dyck reachable path exists between s and t.

The following explains how to solve this example using EUF^n encoding. First, we define a variable n to represent the number of loop executions in the path, and then proceed with the encoding for D_α, resulting in the following:

$$s = f(v_{1\_\text{in}}) \wedge v_{1\_\text{out}} = f^n(v_{1\_\text{in}}) \wedge v_2 = v_{1\_\text{out}} \wedge t = v_2$$

The condition for s and t to be equivalent when processing this formula is $n = 1$, meaning the path that executes the loop once is reachable via D_α. Then proceed with the encoding for D_β, resulting in the following:

$$s' = v'_{1\_\text{in}} \wedge v'_{1\_\text{in}} = g^n(v'_{1\_\text{out}}) \wedge v'_{1\_\text{out}} = g(v'_2) \wedge t' = g(v'_2)$$

The condition for s and t to be equivalent when processing the formula is $n = 0$, meaning the path that does not enter the loop is reachable via D_β. By combining D_α and D_β, the formula has no solution, indicating that there is no path between s and t that is reachable via both D_α and D_β.

An improved version of the SPDS method is the mutual refinement algorithm [Conrado and Pavlogiannis 2024; Ding and Zhang 2023], which removes edges that do not contribute to the current round of Dyck reachability solving. However, such approaches do not fundamentally improve upon the SPDS method; thus, counterexamples satisfying the constraints of Theorem 7 but still causing false positives can be constructed.

LCL-reachability [Zhang and Su 2017] A Linear Conjunctive Language (LCL) describes the language formed by the intersection of linear context-free languages, which provide a precise formulation of Interleaved Dyck languages. However, its reachability solving algorithm is based on finite summaries, which can lead to imprecision due to the limitations of finite summaries and the inability to distinguish different paths between nodes. For example, if the paths between two nodes s and t include $p_1 a$ and $b p_2$, the summary for s and t will be derived from the summaries of p_1 and p_2, even if p_1 and p_2 may not on the same path. In the following example, $\Sigma_\alpha = \{(\!|_1,)\!)_1\}$, $\Sigma_\beta = \{[\![_1,]\!]_1\}$ and the path expression between nodes s and t is:

$$(\!|_1 [\![_1 (\!)_1]\!]_1)^*]\!]_1$$

there exists a path $p_1 \cdot]\!]_1$ where $p_1 = (\!|_1 [\![_1)\!|_1]\!]_1$, and a path $(\!|_1 \cdot p_2$ where $p_2 = [\![_1]\!]_1$. Both p_1 and p_2 are feasible paths, and their summaries are the same, indicating the existence of an interleaved Dyck reachable path. However, according to the summary generation rules of LCL-reachability algorithm [Zhang and Su 2017], two reachable summaries also generate a summary indicating reachability. In this case, there is no interleaved Dyck reachable path between s and s, so LCL-reachability algorithm draws an incorrect conclusion.

The following explains how to solve this example using EUF^n. Similarly, we first define a variable n to represent the number of loop executions, and then proceed with the encoding for D_α, resulting in the following:

$$s = f(v_1) \wedge v_1 = v_{2\_\text{in}} \wedge v_{2\_\text{out}} = f^n(v_{2\_\text{in}}) \wedge t = v_{2\_\text{out}}.$$

By solving the formula, we obtain that s and t are equivalent when $n = 1$. And then proceed with the encoding for D_β, resulting in the following:

$$s' = v'_1 \wedge v'_1 = g(v'_{2\_\text{in}}) \wedge v'_{2\_\text{out}} = g^n(v'_{2\_\text{in}}) \wedge t' = g(v'_{2\_\text{out}}).$$

By solving the formula, we obtain that s and t are equivalent when $n = 0$. By combining $n = 1$ and $n = 0$, we obtain no solution, thus concluding that there is no interleaved Dyck reachable path between s and t.

ILP-based [Li et al. 2023] When solving the single-source single-target IDRP, [Li et al. 2023] derived path expressions between nodes and used ILP to count the number of parentheses and distinguish paths with different loop executions. However, their approach only considered the Parikh images of the corresponding languages and could not encode the relative order of parentheses. Although [Li et al. 2023] used LCL reachability to check the order of parentheses, as mentioned earlier, LCL reachability cannot distinguish between different paths. Therefore, this check cannot fully prevent errors in parentheses order.

For example, given a Dyck reachability instance where $\Sigma = \{(\!|_1, (\!|_2,)\!|_1,)\!|_2\}$, the path expression between nodes s and t is:

$$(\!|_2 ((\!|_1)^*)\!|_2)\!|_1$$

In this case, the ILP method assigns a variable n to represent the number of loop executions and requires that the counts of $(\!|_1$ and $)\!|_1$ be equal, resulting in the constraint $n = 1$. Similarly, it requires the counts of $(\!|_2$ and $)\!|_2$ to be equal, resulting in the constraint $1 = 1$, which is satisfiable, yielding $n = 1$. However, the corresponding path's parentheses order is incorrect, $(\!|_1$ cannot match with $)\!|_2$. Since LCL cannot distinguish between different paths between nodes, it will generate a valid summary as long as there exists any path with a valid parentheses order (*e.g.*, $)\!|_2)\!|_1$). As a result, LCL fails to identify the unique path with matching parentheses counts and incorrectly reports that s and t are reachable, despite the invalid parentheses order.

The following explains how to solve this example using EUF^n. First, assign a loop execution count n to the loop, and then proceed with the encoding for D_α, resulting in the following:

$$s = f_2(v_{1\_\text{in}}) \wedge v_{1\_\text{in}} = f_1^n(v_{1\_\text{out}}) \wedge v_2 = f_2(v_{1\_\text{out}}) \wedge t = f_1(v_2).$$

By solving the formula, we find that no conditional equivalence edge exists between s and t, thus concluding that there is no interleaved Dyck reachable path between them.

4.2 Verification of Uninterpreted Programs

In this subsection, we characterize the class of uninterpreted programs that are encodable in EUF^n and, on this basis, identify a decidable subclass from the perspective of solving EUF^n formulas.

4.2.1 Background of Uninterpreted Program Verification. Uninterpreted programs [Mathur et al. 2019a] are a class of programs that work over infinite domains and relate closely to EUF. Functions in these programs are represented solely by function symbols, serving as abstractions for actual programs [Bryant et al. 2002; Burch and Dill 1994; Fan and He 2024; Lee and Sakallah 2014], enabling quick verification of program properties. Despite the abstraction, the verification of uninterpreted programs remains undecidable [Mathur et al. 2019a]. Recently, a decidable subclass known as *coherent* programs [Mathur et al. 2019a] has been identified, which is applicable to verifying heap manipulation programs [Mathur et al. 2019b]. However, very few real-world programs meet the stringent *coherence* requirements.

Syntax of Uninterpreted Program. Let Σ_{UP} be a finite symbolic set, $\Sigma_f \subseteq \Sigma_{UP}$ represents function set, $\Sigma_V \subseteq \Sigma_{UP}$ represents variable set. The syntax definition of the uninterpreted program is shown in Figure 7, where $\mathtt{f} \in \Sigma_f$ denotes a function, $\mathtt{x}, \mathtt{y} \in \Sigma_V$ denotes a variable, and $\vec{\mathtt{z}}$ is a tuple of variables.

In the above definition, a `skip` statement performs no operation and proceeds directly to the next statement. The program includes two types of assignment statements: variable assignment and function assignment. As well as assume statements, assert statements, and three program structures: sequential structure, `if`-branch structure, and `While`-loop structure. Conditional statements include equivalence relations, as well as corresponding conjunctions and negations. Definitions of the semantics and verification problem for uninterpreted programs can be found in the full version [Du et al. 2026a].

```
stmt  ::=  skip | x := y | x := f(z⃗)
        |  assume(cond) | assert(cond)
        |  stmt; stmt
        |  if (cond) then {stmt;} else {stmt;}
        |  while (cond) {stmt;}
cond  ::=  x = y | cond ∧ cond | ¬cond
```

Fig. 7. The syntax of uninterpreted programs.

4.2.2 An Example of Verifying Uninterpreted Programs Using EUF^n. EUF^n can be used to encode verification problems for partially uninterpreted programs, including some that do not satisfy the *coherence* requirement. We now present an example of verifying a *non-coherent* program using EUF^n. As shown in Figure 8, the goal is to verify whether the assertion at the end of the program holds. The program contains two loops with identical behavior, they can be seen as two versions of the same code, differing only by variable renaming—specifically, `x` to `y` and `z` to `k`. The assertion at the end of the program expresses the functional equivalence of these two code segments. However, since terms computed in the first loop are recomputed in the second without being stored in variables, the program is *non-coherent* [Mathur et al. 2019a] and cannot be verified using the methods in [Mathur et al. 2019a].

We now encode the program using EUF^n. First, for each program variable, we introduce a formula variable to represent its initial value, such as x_0, y_0, z_0, and k_0. Next, for assignment statements, the left-hand variable is rewritten, so we introduce new formula variables to represent the modified values. For example, for the statement `x := y; z := k`, we encode it as $x_1 = y_0 \wedge z_1 = k_0$. We encode the loops in the example program by introducing a natural number variable n for the number of iterations. The encoding consists of three parts: (1) **Entry conditions**: Introduce a universal quantifier i for the i-th entry condition. The condition `x! = z` can be encoded as $i < n \rightarrow next^i(x_1) \neq add^i(z_1)$. (2) **Loop summary**: In the first loop, the statements `x := next(x); z = add(z)` are executed n times, yielding $x_2 = next^n(x_1)$ and $z_2 = add^n(z_1)$. (3) **Exit condition**: After n iterations, when the loop exits and the loop condition is not satisfied, we encode it as $x_2 = z_2$. The encoding for the second loop is similar.

```
x := y;
z := k;
while (x! = z){
  x := next(x);
  z := add(z);
}
while (y! = k){
  y := next(y);
  k := add(k);
}
assert(x = y)
```

$$
\begin{aligned}
&\forall i \forall j. \\
&x_1 = y_0 \\
&\wedge\, z_1 = k_0 \\
&\wedge\, [i < n \rightarrow next^i(x_1) \neq add^i(z_1)] \\
&\wedge\, x_2 = next^n(x_1) \wedge z_2 = add^n(z_1) \\
&\wedge\, x_2 = z_2 \\
&\wedge\, [j < m \rightarrow next^j(y_0) \neq add^j(k_0)] \\
&\wedge\, y_1 = next^m(y_0) \wedge k_1 = add^m(k_0) \\
&\wedge\, y_1 = k_1 \\
&\wedge\, x_2 \neq y_1
\end{aligned}
$$

Fig. 8. An example program and the formula encoding by EUF^n.

This example program lacks branching structures. For programs with branches, we can encode each branch according to the rules for sequential statements and take the disjunction of these encodings. The assertion statement `assert(x = y)` encodes the variables x and y as x_2 and y_1, resulting in the formula $x_2 = y_1$. Thus, this EUF^n formula encodes the existence of a trace that violates the assertion. If the formula is satisfiable, the values of n and m correspond to such a violating trace, suggesting the program is incorrect. Conversely, if unsatisfiable, it indicates no violating traces, confirming the program's correctness.

As previously mentioned, encoding loop conditions introduces formulas of the form $\forall i.i < n \rightarrow next^i(x_1) \neq add^i(z_1)$, where the bounded universal quantifier $\forall i.i < n$ is not supported in the syntax of EUF^n defined in Figure 2. However, a simple modification to the *CCG* algorithm suffices to solve such formulas. For instance, the formula $\forall i.i < n \rightarrow next^i(x_1) \neq add^i(z_1)$ can be expressed in *CCG* as the equivalence of nodes $next^i(x_1)$ and $add^i(z_1)$ under the condition $\forall i.i < n$. Therefore, the EUF^n formula with universal quantifiers in Figure 8 can be transformed into a quantified Presburger arithmetic formula:

$$
\begin{aligned}
\forall i, j. \neg[(i < n \wedge i = m) \vee (i < n \wedge i = n)] \\
\wedge\, \neg[(j < m \wedge j = m) \vee (j < m \wedge j = n)] \wedge \neg(m = n).
\end{aligned}
$$

The unsatisfiability of this formula formally establishes the correctness of the original program. Notably, this program fails to preserve intermediate computation results (specifically, terms computed in the first loop are redundantly recalculated in the second loop without variable retention), thereby violating the coherence criterion. Consequently, such programs cannot be verified as correct under the methodology proposed by Mathur *et al.* [Mathur et al. 2019a].

4.2.3 Decidability of Uninterpreted Programs. We now present a class of uninterpreted programs whose verification problems can be encoded as EUF^n formulas (Theorem 8), and then, based on this class, we identify a decidable subclass of uninterpreted programs (Theorem 9).

Theorem 8 (EUF^n Encodability of Uninterpreted Programs). The verification problem for uninterpreted programs without nested loops, conditional branches within loops, or multi-ary functions in loop bodies can be encoded as the satisfiability problem of EUF^n formulas.

Proof. The detailed proof is provided in the full version [Du et al. 2026a]. □

Theorem 9 (Decidability of the Uninterpreted Programs Verification Problem). There exists a decidable class $\mathcal{P}$ of uninterpreted programs where: (1) Every program $P \in \mathcal{P}$ can be encoded into an EUF^n formula ϕ_P; (2) For any assume statement of the form `assume(x = y)` in P, the superterm edges set $E_{\succ}$ contains no element e satisfying $e = (x, \phi, y)$ with ϕ is satisfiable.

Proof. Encoding loops in uninterpreted programs introduces universal quantifiers. According to Algorithm 1 and the discussion in Section 3.5, if **periodic equivalence** relations do not appear in the formula, then no divisibility predicates are introduced in the conditions of equivalence edges.

Theorem 9, specifically Condition (2), ensures the absence of such **periodic equivalence** relations. As a result, the EUF^n formula obtained from encoding this class of uninterpreted programs can be transformed into a quantified Presburger arithmetic formula. The satisfiability of the original formula is preserved under this transformation by Algorithm 1, and the resulting formula is decidable. Therefore, the verification problem for this class of uninterpreted programs is decidable. □

For any given uninterpreted program, determining whether it satisfies Condition (1) is straightforward. After encoding, Condition (2) can be verified during the *CCG*-based solving process. The decidable subclass of uninterpreted programs established in Theorem 9 has a non-empty intersection with the existing class of coherent decidable programs [Mathur et al. 2019a,b], but neither class subsumes the other. For example, Figure 8 illustrates a non-coherent program; on the other hand, uninterpreted programs with nested loop structures, while potentially coherent, cannot be encoded in EUF^n. Furthermore, [Gulwani and Tiwari 2007] abstract programs using uninterpreted functions, where conditional branches are replaced by nondeterministic choice, equality guards are omitted, and the verification mainly targets equality assertions. [Godoy and Tiwari 2009] proposed a decidable class of uninterpreted programs called Sloopy programs, which similarly abstracts control flow with nondeterministic choice and reduces assertion conditions to Presburger formulas, an idea closely related to the verification approach in this paper. The decidable classes in these two works both have a non-empty intersection with the class defined in this paper, but neither is subsumed by the other. Their classes support multi-argument uninterpreted functions inside loops, whereas our approach can directly model branch conditions in programs.

The limitations of EUF^n in encoding nested loops and branching within loops stem from the absence of variables representing inner iteration counts and the inability to handle compositions of multi-arity functions at *parametric depth*, thereby restricting loops to unary functions. Despite this, this class of uninterpreted programs remains practically useful: it can model heap operations (e.g., reads/writes [Mathur et al. 2019b]), and multi-arity functions can be abstracted into unary functions with vector arguments, thereby falling within this class. Moreover, in control property verification, function semantics are often irrelevant in that the order and frequency of specific operations are what truly matter [Bueno 2021]. For instance, in the OpenSSL vulnerability CVE-2015-0291, a deallocated state variable `shared_sigalgs` was not synchronized with its length field `shared_sigalgslen`, leading to a null dereference. Such bugs can be abstracted as constraints on update counts of related variables, *e.g.*, using counter variables with updates like $\texttt{count}_\texttt{i} = \texttt{add}(\texttt{count}_\texttt{i})$, and enforcing relations such as $count_i = count_j$ or $count_i = add^n(count_j) \wedge n > 0$.

5 Related Work

The EUF theory is widely applied in program verification, such as in verifying pipeline processor equivalence [Burch and Dill 1994] and regression verification for C programs [Strichman and Godlin 2005]. The satisfiability of its quantifier-free fragment is decidable. Early algorithms by Shostak [Shostak 1984], Nelson and Oppen [Nelson and Oppen 1980], and Downey *et al.* [Downey et al. 1980] leverage directed acyclic graph (DAG) and *Union-Find* data structure to achieve $O(n \log n)$ congruence closure complexity. Bachmair *et al.* [Bachmair and Tiwari 2000; Bachmair et al. 2003] later unified these algorithms via a rewriting framework. Nieuwenhuis and Oliveras [Nieuwenhuis and Oliveras 2005, 2007] introduced an incremental algorithm with an *Explain* feature, now standard in SMT solvers, which outputs a small subset of constraints implying term equivalence. Congruence closure algorithms are also vital in automated theorem proving [De Moura and Bjørner 2007] and compiler optimization [Tate et al. 2009]. The Egg library [Willsey et al. 2021], implementing equality

saturation [Tate et al. 2009], enhances performance via a *rebuilding* mechanism that amortizes consistency maintenance costs.

In extending EUF, Bryant *et al.* [Bryant et al. 1999] introduced Positive EUF (PEUF), which defines positive terms. While PEUF does not increase EUF's expressive power, it significantly improves solving efficiency. Additionally, Bryant *et al.* [Bryant et al. 2002] proposed CLU (Counter arithmetic with Lambda expressions and Uninterpreted functions), which extends EUF by allowing integer-valued domains for functions and predicates. This extension is orthogonal to our work, which introduces *parametric composition depth* for unary functions. Additionally, Zakhour *et al.* [Zakhour et al. 2025] extended e-graphs (an implementation of congruence graphs) with a unified framework for handling both equalities and disequalities. Deepak [Deepak 2019] introduced conditional congruence closure using Horn equations, where target equalities hold only under conditions expressed as conjunctions of EUF equalities.

On the other hand, the satisfiability definition of EUF^n formulas (Definition 1) implies that enumerating all assignments to the natural number variables yields an infinite set of EUF formulas, with the EUF^n formula being satisfiable if and only if at least one formula in this set is satisfiable. This is closely related to the concept of *disjunctive satisfiability* introduced by Mathur *et al.* [Mathur et al. 2025], which studies the existence of a model satisfying at least one formula in a given infinite formula set. While their work focuses on regular first-order theories with formula sets described by regular tree languages, EUF^n can express non-regular terms such as $f^n(g^n(h^n(x)))$, which cannot be captured by regular tree languages. Thus, our work addresses the *disjunctive satisfiability* of a non-regular EUF theory, extending beyond the regular framework of [Mathur et al. 2025].

6 Conclusion

This paper introduces EUF^n, a decidable extension of EUF theory that supports *parametric composition depth* for unary functions (*e.g.*, $f^n(x)$), significantly enhancing the theory's expressive power. We also present a *CCG* algorithm designed to solve quantifier-free EUF^n formulas by reducing them to existential sentence of Presburger arithmetic with divisibility. This satisfiability-preserving transformation establishes the decidability of the satisfiability of quantifier-free EUF^n formulas, as the corresponding Presburger fragment is known to be decidable. Furthermore, we demonstrate the applications of EUF^n through two practical scenarios: the single-source single-target IDRP and the verification of uninterpreted programs, for which we identify novel decidable subclasses.

Acknowledgments

This research was supported by the National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Program (No. 62172429 and U2341212). We also thank the anonymous reviewers for their valuable feedback.

A Axioms of Presburger Arithmetic [Mendelson 2009]

Let the language $\mathcal{A}_0 = \{0, S, +\}$, where 0 is a constant symbol, S is a unary function symbol, $+$ is a binary function symbol. The axioms of Presburger arithmetic are defined as the universal closures of the following formulas:

(1) $x = y \rightarrow (x = z \rightarrow y = z)$
(2) $x = y \rightarrow S(x) = S(y)$
(3) $\neg(S(x) = 0)$
(4) $S(x) = S(y) \rightarrow x = y$
(5) $x + 0 = x$
(6) $x + S(y) = S(x + y)$

(7) The above axioms define the properties of equality (axioms (1) and (2)), the successor function, and the addition function. Additionally, Presburger arithmetic includes an infinite axiom schema, the induction axiom: Let $P(x)$ be a first-order formula in the language $\mathcal{A}_0$, where x is a free variable of P. The induction axiom is formulated as follows:

$$(P(0) \wedge \forall x\,(P(x) \rightarrow P(S(x)))) \rightarrow \forall y\,P(y).$$

The less-than predicate $<$ is definable in the language $\mathcal{A}_0$ by the formula $x < y \Leftrightarrow \exists z\,(\neg(z = 0) \wedge x + z = y)$.

B Details of the proof of Lemma 1.

Lemma B.1. *(Lemma 1) During Algorithm 1, for every equivalence edge $(term_l, \phi, term_r)$ in the* CCG *and any assignment $\sigma : \mathsf{N} \rightarrow \mathbb{N}$, the following invariant holds: ϕ is satisfiable under σ if and only if $term_l$ and $term_r$ belong to the same equivalence class in the congruence closure of $\Psi[\sigma]$.*

Proof. Given a conjunct of an EUF^n formula, we prove by induction that after processing each equation, the *CCG* and the congruence closure of $\Psi[\sigma]$ obtained at that step satisfy the invariant stated in Lemma 1.

- **Base Step**
  The initialization rules (Section 3.3) show that equivalence edges in $E_{\equiv}$ are of the form $(term_1, k_1 = k_2, term_2)$, where $term_1 = f^{k_1}(term)$ and $term_2 = f^{k_2}(term)$. When $\sigma(k_1) = \sigma(k_2)$, the terms $term_1$ and $term_2$ are identical and thus belong to the same equivalence class in the congruence closure of $\Psi[\sigma]$. Conversely, in the EUF formula $\Psi[\sigma]$ obtained under any assignment σ, terms of the form $f^c(term)$ (where c is a constant) are treated identically during initialization. Setting $k_1 = c$ and $k_2 = c$ yields a satisfying assignment for $k_1 = k_2$.
- **Inductive Step**

  Assume the invariant of Lemma 1 holds after the first k equations. For the $(k{+}1)$-th equation, the algorithm adds equivalence edges via `EqProp`, which may be invoked multiple times per equation. We therefore prove by induction on the number of `EqProp` executions that the invariant still holds after processing the $(k{+}1)$-th equation.
  Initially, when starting the $(k{+}1)$-th equation, the invariant holds by the induction hypothesis. Now assume it holds after the first k' executions of `EqProp` for this equation. When executing the $(k'{+}1)$-th `EqProp`, it adds equivalence edges to the *CCG* in either Line 4 or Line 9. In Line 20, to maintain transitivity of the equivalence relation, `AddEq` adds the equivalence edge $(term_1, \phi_{\text{old}} \vee (\phi_{\text{new}} \wedge \phi_1 \wedge \phi_2), term_2)$ to the *CCG* (from Lines 20, 22, and 23), where $term_1 \in [term_l]$ and $term_2 \in [term_r]$. By the induction hypothesis, the edges $(term_l, \phi_1, term_1)$, $(term_r, \phi_2, term_2)$, and $(term_1, \phi_{\text{old}}, term_2)$ satisfy the invariant of Lemma 1. As established previously, $(term_l, \phi_{\text{new}}, term_r)$ also satisfies the invariant of Lemma 1 (*i.e.*, the two parameter sources of `EqProp` in Lines 4 and 9). By the transitivity of the equivalence relation, $term_1$ and $term_2$ are also equivalent under $\phi_{\text{new}} \wedge \phi_1 \wedge \phi_2$. Hence, the updated edge $(term_1, \phi_{\text{old}} \vee (\phi_{\text{new}} \wedge \phi_1 \wedge \phi_2), term_2)$ satisfies the invariant of Lemma 1.

□

C Details of the proof of Lemma 2 and Theorem 4.

Lemma C.1. *(Lemma 2) Assume $n > m$, $term = f^m(term) = f^n(term) \leftrightarrow term = f^{\gcd(m,n)}(term)$.*

Proof. As the implication from right to left is obvious, we now prove the implication from left to right, *i.e.*, $term = f^m(term) = f^n(term) \rightarrow term = f^{\gcd(m,n)}(term)$. Here we assume $m < n$.

By Theorem 1, the identity $term = f^m(term)$ implies $f^{n-m}(term) = f^n(term)$, and thus by the transitivity of the equivalence relation:

$$term = f^{n-m}(term) = f^m(term) = f^n(term)$$

If $n - m \neq m$, we can can repeatedly apply Theorem 1 until $term = f^a(term) = f^b(term)$ $s.t.$ $a = b$ is derived. This process utilizes a variant of the Euclidean algorithm [Rosen 2011] to find the greatest common divisor of m and n. Thus, $a = b = \gcd(m, n)$, and the formula on the right $term = f^{\gcd(m,n)}(term)$ is derived. □

Theorem C.2. *(Theorem 4) Given two periodic equivalence relations* $term = f^k(term)$ *and* $f^m(term) = f^n(term)$. *Then*

$$term = f^k(term) \wedge f^m(term) = f^n(term) \leftrightarrow term = f^{\gcd(k,(n-m))}(term).$$

Proof. We will prove the theorem from two directions. Without loss of generality, we assume that $n > m$.

- Right to Left:
  Let $c_1 = \gcd(k, (n - m))$. Since c_1 is a divisor of k, we have

  $$term = f^{c_1}(term) \rightarrow term = f^k(term),$$

  and since c_1 is also a divisor of $(n - m)$, we have

  $$term = f^{c_1}(term) \rightarrow term = f^{(n-m)}(term).$$

  Applying f repeatedly for m times to both sides of the above equation, we obtain

  $$f^m(term) = f^n(term).$$

  Thus, the right-to-left direction is established.
- Left to Right:
  From the formula on the left, we can obtain

  $$term = f^{c_1 k}(term), \quad s.t. c_1 = 0, 1, \cdots,$$

  $$f^{m+c_2}(term) = f^{n+c_2}(term), \quad s.t. c_2 = 0, 1, \cdots.$$

  For any given n and k, when c_1 is sufficiently large, there must exist a $c_2 > 0$ such that $c_1 k = n + c_2$, that is, there exist c_1, c_2 such that

  $$term = f^k(term) = f^{c_1 k}(term) = f^{n+c_2}(term) = f^{m+c_2}(term).$$

  By Lemma 2, it follows that

  $$term = f^{\gcd(k, m+c_2)}(term).$$

  From the equation $c_1 k = n + c_2$, it follows that this equality can also be expressed as

  $$term = f^{\gcd(k,\, c_1 k - (n-m))}(term).$$

  Since $\gcd(m, n) = \gcd(m, n - m)$ and $\gcd(m, n) = \gcd(m, -n)$, we have

  $$\gcd(k,\ c_1 k - (n - m)) = \gcd(k, n - m).$$

  Hence, the left-to-right direction holds, and the proof is thus complete.

□

References

Leo Bachmair and Ashish Tiwari. 2000. Abstract congruence closure and specializations. In *International Conference on Automated Deduction*. Springer, 64–78. doi:10.1007/10721959_5

Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. 2003. Abstract congruence closure. *Journal of Automated Reasoning* 31, 2 (2003), 129–168. doi:10.1023/b:jars.0000009518.26415.49

Anatoly Petrovich Bel'tyukov. 1980. Decidability of the universal theory of natural numbers with addition and divisibility. *Journal of Soviet Mathematics* 14 (1980), 1390–1404. doi:10.1007/bf01693974

Randal E Bryant, Steven German, and Miroslav N Velev. 1999. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11*. Springer, 470–482. doi:10.1007/3-540-48683-6_40

Randal E Bryant, Shuvendu K Lahiri, and Sanjit A Seshia. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*. Springer, 78–92. doi:10.1007/3-540-45657-0_7

Denis Bueno. 2021. *Software Model Checking with Uninterpreted Functions*. Ph. D. Dissertation.

Jerry R Burch and David L Dill. 1994. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification: 6th International Conference, CAV'94 Stanford, California, USA, June 21–23, 1994 Proceedings 6*. Springer, 68–80. doi:10.1007/3-540-58179-0_44

Giovanna Kobus Conrado, Adam Husted Kjelstrøm, Jaco van de Pol, and Andreas Pavlogiannis. 2025. Program Analysis via Multiple Context Free Language Reachability. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 509–538. doi:10.1145/3704854

Giovanna Kobus Conrado and Andreas Pavlogiannis. 2024. A Better Approximation for Interleaved Dyck Reachability. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 18–25. doi:10.1145/3652588.3663318

Leonardo De Moura and Nikolaj Bjørner. 2007. Efficient E-matching for SMT solvers. In *International Conference on Automated Deduction*. Springer, 183–198. doi:10.1007/978-3-540-73595-3_13

Kapur Deepak. 2019. Conditional congruence closure over uninterpreted and interpreted symbols. *Journal of Systems Science and Complexity* 32, 1 (2019), 317–355. doi:10.1007/s11424-019-8377-8

Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *International Static Analysis Symposium*. Springer, 231–258. doi:10.1007/978-3-031-44245-2_12

Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. 1980. Variations on the common subexpression problem. *Journal of the ACM (JACM)* 27, 4 (1980), 758–771. doi:10.1145/322217.322228

Yide Du, Zhenbang Chen, Weijiang Hong, and Wei Dong. 2026a. EUF^{n}: A Decidable Extension to the Theory of Equality with Uninterpreted Functions. arXiv:2608.21478 [cs.LO] https://arxiv.org/abs/2608.21478

Yide Du, Zhenbang Chen, Kunlin Liu, Guofeng Zhang, Xudong Wang, Ke Ma, Wei Dong, and Ji Wang. 2026b. EUF-based Solving Dyck-Reachability with Applications to Static Analysis. In *International Symposium on Formal Methods*. Springer, 296–316. doi:10.1007/978-3-032-26220-2_15

Hongyu Fan and Fei He. 2024. Leveraging Datapath Propagation in IC3 for Hardware Model Checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024). doi:10.1109/tcad.2024.3360022

Guillem Godoy and Ashish Tiwari. 2009. Invariant checking for programs with procedure calls. In *International Static Analysis Symposium*. Springer, 326–342. doi:10.1007/978-3-642-03237-0_22

VK Hari Govind, Sharon Shoham, and Arie Gurfinkel. 2021. Logical characterization of coherent uninterpreted programs. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 77–85. doi:10.34727/2021/isbn.978-3-85448-046-4_16

Sumit Gulwani and Ashish Tiwari. 2007. Assertion checking unified. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 363–377. doi:10.1007/978-3-540-69738-1_26

Weijiang Hong, Zhenbang Chen, Yide Du, and Ji Wang. 2021. Trace abstraction-based verification for uninterpreted programs. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*. Springer, 545–562. doi:10.1007/978-3-030-90870-6_29

Daniel Kroening and Ofer Strichman. 2008. *Decision procedures*. Vol. 5. Springer. doi:10.1007/978-3-540-74105-3

Antonia Lechner, Joël Ouaknine, and James Worrell. 2015. On the complexity of linear arithmetic with divisibility. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 667–676. doi:10.1109/lics.2015.67

Suho Lee and Karem A Sakallah. 2014. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 849–865. doi:10.1007/978-3-319-08867-9_56

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2023. Single-Source-Single-Target Interleaved-Dyck Reachability via Integer Linear Programming. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1003–1026. doi:10.1145/3571228

Leonard Lipshitz. 1978. The Diophantine problem for addition and divisibility. *Trans. Amer. Math. Soc.* (1978), 271–283. doi:10.1090/s0002-9947-1978-0469886-1

Umang Mathur, P Madhusudan, and Mahesh Viswanathan. 2019a. Decidable verification of uninterpreted programs. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. doi:10.1145/3290359

Umang Mathur, David Mestel, and Mahesh Viswanathan. 2025. The Decision Problem for Regular First Order Theories. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 986–1012. doi:10.1145/3704870

Umang Mathur, Adithya Murali, Paul Krogmeier, P Madhusudan, and Mahesh Viswanathan. 2019b. Deciding memory safety for single-pass heap-manipulating programs. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29. doi:10.1145/3371103

Elliott Mendelson. 2009. *Introduction to mathematical logic.* Chapman and Hall/CRC. doi:10.1201/9781584888772

Markus Müller-Olm, Oliver Rüthing, and Helmut Seidl. 2005. Checking Herbrand equalities and beyond. In *Verification, Model Checking, and Abstract Interpretation: 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005. Proceedings 6.* Springer, 79–96. doi:10.1007/978-3-540-30579-8_6

Greg Nelson and Derek C Oppen. 1980. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)* 27, 2 (1980), 356–364. doi:10.1145/322186.322198

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications.* Springer, 453–468. doi:10.1007/978-3-540-32033-3_33

Robert Nieuwenhuis and Albert Oliveras. 2007. Fast congruence closure and extensions. *Information and Computation* 205, 4 (2007), 557–580. doi:10.1016/j.ic.2006.08.009

Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 162–186. doi:10.1145/345099.345137

Kenneth H Rosen. 2011. *Elementary number theory.* Pearson Education London.

Kenneth H Rosen and Kamala Krithivasan. 1999. *Discrete mathematics and its applications.* Vol. 6. McGraw-hill New York.

Robert E Shostak. 1978. An algorithm for reasoning about equality. *Commun. ACM* 21, 7 (1978), 583–585. doi:10.1145/359545.359570

Robert E Shostak. 1984. Deciding combinations of theories. *Journal of the ACM (JACM)* 31, 1 (1984), 1–12. doi:10.1145/2422.322411

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. doi:10.1145/3290361

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. *ACM SIGPLAN Notices* 40, 10 (2005), 59–76. doi:10.1145/1094811.1094817

Ofer Strichman and Benny Godlin. 2005. Regression verification-A practical way to verify programs. In *Working Conference on Verified Software: Theories, Tools, and Experiments.* Springer, 496–501. doi:10.1007/978-3-540-69149-5_54

Robert Endre Tarjan. 1981. Fast algorithms for solving path problems. *Journal of the ACM (JACM)* 28, 3 (1981), 594–614. doi:10.1145/322261.322273

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 264–276. doi:10.1145/1480881.1480915

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434304

George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2282–2305. doi:10.1145/3704913

Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation.* 435–446. doi:10.1145/2491956.2462159

Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* 344–358. doi:10.1145/3009837.3009848

Received 2026-03-17; accepted 2026-06-10