Single-Source-Single-Target Interleaved-Dyck Reachability via Integer Linear Programming

YUANBO LI, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA
THOMAS REPS, University of Wisconsin–Madison, USA

An interleaved-Dyck (InterDyck) language consists of the interleaving of two or more Dyck languages, where each Dyck language represents a set of strings of balanced parentheses. InterDyck-reachability is a fundamental framework for program analyzers that simultaneously track multiple properly-matched pairs of actions such as call/return, lock/unlock, or write-data/read-data. Existing InterDyck-reachability algorithms are based on the well-known tabulation technique.

This paper presents a new perspective on solving InterDyck-reachability. Our key observation is that for the *single-source-single-target* InterDyck-reachability variant, it is feasible to summarize all paths from the source node to the target node based on *path expressions*. Therefore, InterDyck-reachability becomes an InterDyck-path-recognition problem over path expressions. Instead of computing summary edges as in traditional tabulation algorithms, this new perspective enables us to express InterDyck-reachability as a *parenthesis-counting* problem, which can be naturally formulated via integer linear programming (ILP).

We implemented our ILP-based algorithm and performed extensive evaluations based on two client analyses (a reachability analysis for concurrent programs and a taint analysis). In particular, we evaluated our algorithm against two types of algorithms: (1) the general all-pairs InterDyck-reachability algorithms based on linear conjunctive language (LCL) reachability and synchronized pushdown system (SPDS) reachability, and (2) two domain-specific algorithms for both client analyses. The experimental results are encouraging. Our algorithm achieves 1.42×, 28.24×, and 11.76× speedup for the concurrency-analysis benchmarks compared to all-pair LCL-reachability, SPDS-reachability, and domain-specific tools, respectively; 1.2×, 69.9×, and 0.98× speedup for the taint-analysis benchmarks. Moreover, the algorithm also provides precision improvements, particularly for taint analysis, where it achieves 4.55%, 11.1%, and 6.8% improvement, respectively.

CCS Concepts: • **Theory of computation → Program analysis**; • **Mathematics of computing →** *Graph algorithms.*

Additional Key Words and Phrases: Interleaved-Dyck Reachability, Path Expressions, Integer Linear Programming, Single-Source-Single-Target Variant

ACM Reference Format:
Yuanbo Li, Qirun Zhang, and Thomas Reps. 2023. Single-Source-Single-Target Interleaved-Dyck Reachability via Integer Linear Programming . *Proc. ACM Program. Lang.* 7, POPL, Article 35 (January 2023), 24 pages. https://doi.org/10.1145/3571228

1 INTRODUCTION

Formal-language reachability [Yannakakis 1990] (or L-reachability) is a popular framework to formulate many static-analysis applications [Reps 1998]. An L-reachability instance consists of

Authors' addresses: Yuanbo Li, Georgia Institute of Technology, Atlanta, USA, yuanboli@gatech.edu; Qirun Zhang, Georgia Institute of Technology, Atlanta, USA, qrzhang@gatech.edu; Thomas Reps, University of Wisconsin–Madison, Madison, USA, reps@cs.wisc.edu.

2475-1421/2023/1-ART35
https://doi.org/10.1145/3571228

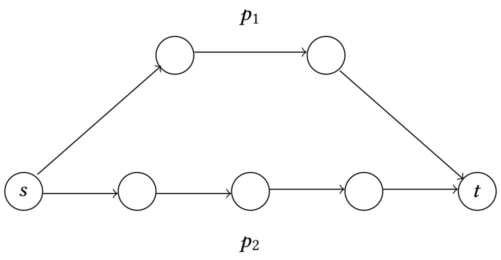


Fig. 1. Multi-path imprecision in SPDS-reachability and LCL-reachability algorithms. This figure is a generalization of the example discussed in Zhang and Su [2017, Fig. 5].

(1) an edge-labeled graph representing the program under analysis, and (2) a formal language L that captures the program properties to be analyzed. Two nodes are considered reachable in the L-reachability instance if they are joined by a path, and the realized string of the path belongs to the formal language L. Perhaps the best-known L-reachability formulation in program analysis is Dyck-reachability [Kodumal and Aiken 2004; Reps 1998; Zhang et al. 2013]. Specifically, it can be used to model matched-parenthesis program properties, such as function calls/returns [Reps 2000; Reps et al. 1995], field reads/writes [Huang et al. 2015; Zhang et al. 2013], pointer references/dereferences [Zhang et al. 2014; Zheng and Rugina 2008].

Interleaved-Dyck (InterDyck) reachability [Reps 2000] is a natural extension to Dyck-reachability by simultaneously tracking multiple matched-parenthesis program properties. InterDyck-reachability is more expressive, which can formulate more complicated analysis problems, such as context-sensitive data-dependence analysis [Zhang and Su 2017]. However, InterDyck-reachability is undecidable [Reps 2000], and hence practical InterDyck-reachability algorithms are based on approximations (and are inherently imprecise). Existing approaches for solving InterDyck-reachability include the traditional context-free-language (CFL) reachability [Reps 1998], synchronized pushdown system (SPDS) reachability [Späth et al. 2019], and linear-conjunctive-language (LCL) reachability [Zhang and Su 2017]. These InterDyck-reachability algorithms are based on tabulation [Reps 1998]. Specifically, they compute a succinct representation of paths between node pairs called *summaries* to derive the reachability results. These algorithms typically over-approximate InterDyck-reachability in two ways: (1) The first one is by utilizing path over-approximations, *i.e.*, the algorithms treat all the paths between two nodes as if there is only one single path. Specifically, in SPDS-reachability and LCL-reachability algorithms, properties in a single path between a pair of nodes are considered to exist along all paths between the nodes. (2) The second way is by utilizing over-approximative summaries, *i.e.*, the summaries themselves cannot precisely express InterDyck-reachability. For example, the CFL-reachability approach for InterDyck-reachability [Huang et al. 2015; Reps 1998; Yan et al. 2011] uses over-approximative summaries because the InterDyck language is not a CFL. Specifically, one of the Dyck languages in the InterDyck language is over-approximated using a regular language. Then the InterDyck-reachability problem becomes a CFL-reachability problem, which can be solved by the standard CFL-reachability tabulation algorithm [Reps 1998].

The two over-approximation methods introduce the following sources of imprecision:

- *Multi-path imprecision.* An InterDyck-reachability algorithm using path over-approximation cannot distinguish different paths between a pair of nodes. For example, in the LCL-reachability and SPDS-reachability algorithms, an InterDyck-path needs to satisfy two (or more) summary conditions expressed in the corresponding Dyck languages simultaneously.

However, the algorithms can check only one summary condition at a time. They cannot distinguish different paths between a pair of nodes, and conservatively aggregate the summaries of all paths, which leads to imprecision. Consider the example in Figure 1. Suppose that there exists a path p_1 between node s and node t satisfying one summary condition A, and another path p_2 satisfying summary condition B. The algorithms conservatively assume that there exists a path between s and t that satisfies A and B simultaneously.

- *Finite summary approximation.* Another source of imprecision comes from the imprecise formulation based on finite summaries. For example, the CFL-reachability approach uses CFL summaries to approximate InterDyck-reachability. These finite summaries are inherently approximations and thus introduce imprecision.

The two over-approximation techniques are indeed pervasive [Arzt et al. 2014; Feng et al. 2015; Späth et al. 2019; Zhang and Su 2017], and almost all existing *all-pairs* InterDyck-reachability algorithms utilize at least one of the techniques. The key enabling insight of our approach is that for *single-source-single-target* InterDyck-reachability, it is unnecessary to build summaries for all intermediate node pairs that belong to the paths from the source s to the target t. Specifically, we can explicitly represent all paths from s to t based on *path expressions* [Tarjan 1981]. Over path expressions, InterDyck-reachability becomes a *parenthesis-counting problem.* We can formulate the parenthesis-counting problem as an integer linear programming (ILP) problem [Schrijver 1998], which can be effectively solved by off-the-shelf ILP solvers [Gurobi Optimization, LLC 2021]. To sum up, the essence of our approach consists of two principal ideas, which mitigate, respectively, the corresponding sources of imprecision in traditional summary-based tabulation algorithms.

- *Path expressions.* To avoid multi-path imprecision, instead of tracking reachability between node pairs, our proposed algorithm computes path expressions, which represent all paths between nodes s and t, and transforms the InterDyck-reachability problem into a valid-path-recognition problem. With this new perspective, for the same example in Figure 1, the algorithm analyzes paths p_1 and p_2 independently, avoiding the multi-path imprecision.
- *Parenthesis-counting.* We address the finite summary approximation by transforming the reachability problem into a parenthesis-counting problem based on an ILP formulation over path expressions. A path expression provides the structure of all paths between the source and the target. In a path expression, each Kleene star represents some number of traversals of a cycle. Along a valid InterDyck-path, for each kind of parentheses, there should always be more open parentheses than close parentheses. We use these restrictions to create an ILP encoding and check whether there exists a valid path after completing a certain number of traversals of a cycle. Indeed, the ILP solver explores an infinite set of natural numbers as the search space for each cycle. Consequently, it can explore every possible path but avoid using a finite number of summaries in reachability computation.

Single-source-single-target InterDyck-reachability is particularly useful for formulating demand-driven program-analysis problems [Heintze and Tardieu 2001; Sridharan et al. 2005; Yan et al. 2011]. We implemented our single-source-single-target InterDyck-reachability algorithm and evaluated it using two client analyses: a concurrent program state-reachability analysis on the DDVerify benchmark from Witkowski et al. [2007], and a demand-driven taint analysis on the benchmark from Huang et al. [2015]. Table 1 summarizes the experimental results. Specifically, we compared our algorithm against two state-of-the-art InterDyck-reachability algorithms, SPDS-reachability and LCL-reachability. In addition, we also compared it with two domain-specific analysis tools. For concurrent program analysis, we compared our algorithm with CUBA [Liu and Wahl 2018], a context-unbounded analysis tool for recursive concurrent programs. For the taint-analysis application, we compared our algorithm with the DroidInfer tool [Huang et al. 2015].

Table 1. Summary of our improvement results in experiments. For the concurrency-analysis benchmark, because the domain-specific tool CUBA utilizes under-approximations, we use the notation "-" to represent the precision is incomparable for two algorithms.

	vs. LCL-reachability			vs. SPDS-reachability			vs. Domain-specific	
	Speedup	Precision	Memory	Speedup	Precision	Memory	Speedup	Precision
Concurrency	1.42×	0%	57%	28.24×	6.8%	-10.4%	11.76×	-
Taint analysis	1.20×	4.55%	53%	69.9×	11.1%	59.4%	0.98×	6.8%

The experimental results are encouraging: our proposed algorithm achieves better performance and precision results. Specifically, as shown in columns 2, 5, and 8 in Table 1, our algorithm achieves 1.42×, 28.24×, and 11.76× speedup for the concurrency-analysis benchmark compared to all-pairs LCL-reachability, SPDS-reachability, and domain-specific tools, respectively; 1.20×, 69.9×, and 0.98× speedup for the taint-analysis benchmark. Moreover, for the demand-driven queries, columns 3, 6, and 9 show that our algorithm improves the analysis precision, particularly for taint analysis, where it achieves 4.55%, 11.1%, and 6.8% improvement, respectively. Columns 4 and 7 demonstrate the memory consumption results. Our algorithm can save more than 50% in comparison with SPDS-reachability in the taint-analysis benchmark, and both LCL-reachability and SPDS-reachability in the concurrency-analysis benchmark.

The contributions of the work are as follows:

- We present a novel single-source-single-target InterDyck-reachability algorithm. Our algorithm is based on a new technical insight: using path expressions and ILP to address two sources of imprecision in the state-of-the-art InterDyck-algorithms.
- We implemented and evaluated the algorithm against two state-of-the-art InterDyck-reachability algorithms and two domain-specific static analysis tools. The experimental results demonstrate that: (i) our algorithm provides significant speedup and better precision compared to the all-pairs InterDyck-reachability algorithms; and (ii) it shows more than an order-of-magnitude of speedup compared with the context-unbounded analysis tool CUBA.

Organization. The remainder of the paper is organized as follows: Section 2 discusses a motivating example that illustrates the benefits of our algorithm. Section 3 introduces terminology and notations. Section 4 presents the details of the algorithm. Section 5 presents the experimental results. Section 6 discusses related work. Section 7 concludes.

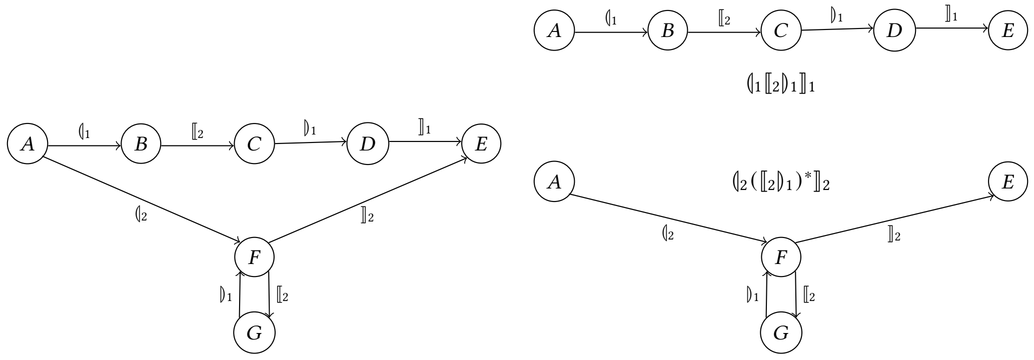
2 MOTIVATING EXAMPLE

This section motivates our single-source-single-target InterDyck-reachability algorithm using a concrete example in Figure 2. Consider the InterDyck-reachability between nodes A and E in Figure 2a. It is straightforward to see that there is no InterDyck-path between them. However, the state-of-the-art SPDS-reachability and LCL-reachability algorithms both report that E is InterDyck-reachable from A. We illustrate how multi-path imprecision and finite-state approximation occur in those two algorithms, and then describe how our algorithm addresses the imprecision.

2.1 Existing InterDyck-Reachability Algorithms

We first briefly introduce the state-of-the-art InterDyck-reachability algorithms.

- *SPDS-reachability* [Späth et al. 2019]. The SPDS-reachability algorithm utilizes a synchronized pushdown system (SPDS) to model InterDyck-reachability. Specifically, the SPDS model employs stacks to describe the Dyck languages in the InterDyck language. The reachability



(a) Input graph. (b) Path expressions and their corresponding paths.

Fig. 2. Computing single-source-single-target InterDyck-reachability between node A and node E.

algorithm runs in two phases for the InterDyck language of two Dyck languages. In each phase, the SPDS-reachability algorithm uses one stack to precisely model one of the Dyck languages while discarding the other Dyck language. If a node pair is Dyck-reachable in both phases, it is considered to be InterDyck-reachable.

- *LCL-reachability* [Zhang and Su 2017]. The LCL-reachability algorithm reformulates the InterDyck-reachability problem as an equivalent linear-conjunctive-language reachability problem. The algorithm over-approximates LCL-reachability by computing summaries based on the tabulation technique. The summary of a realized string $s \in \Sigma^*$ can be determined by the summaries of its sub-strings $s_1, s_2 \in \Sigma^*$, where $s = s_1 \cdot b = a \cdot s_2$ with $a, b \in \Sigma$ and "$\cdot$" denotes string concatenation operation.

Both algorithms report a false-positive result that node E is InterDyck-reachable from node A in Figure 2a. We then discuss the two sources of imprecision.

- *Imprecision in SPDS-reachability.* The multi-path imprecision in the SDPS-reachability algorithm stems from the two-phase reachability computation described above. For each phase, the algorithm recognizes a set of reachable node pairs for one Dyck-reachability problem. If a node pair is reachable in both phases, it is possible that the node pair is reachable via two different paths, and there exist no InterDyck-paths. Consider the motivating example in Figure 2. The path $A \longrightarrow B \longrightarrow C \longrightarrow D \longrightarrow E$ is a Dyck-path of the parenthesis language if the brackets along the path are ignored. Similarly, nodes A and E are Dyck-reachable in the path $A \longrightarrow F \longrightarrow G \longrightarrow F \longrightarrow E$ for the bracket Dyck language (when parentheses are ignored). The SPDS-reachability algorithm recognizes the node pair (A, E) in both Dyck-reachability problems—and thus it reports the node pair (A, E) as InterDyck-reachable—even though Dyck-reachability for the two Dyck languages holds along two different paths.
- *Imprecision in LCL-reachability.* The LCL-reachability algorithm exhibits both multi-path imprecision and finite-summary approximation in this example. The LCL-reachability algorithm uses finite summaries to model InterDyck-reachability, which is inherently approximative. For example, LCL-reachability represents path strings "$⦇_1$" and "$⦇_1⦇_1$" using the same summary. In addition, the LCL-reachability algorithm exhibits multi-path imprecision as discussed by Zhang and Su [2017, Sec. 7.2]. Recall that to obtain the summary of a string s in LCL-reachability, we need the summaries of s_1 and s_2 where $s = s_1 \cdot b = a \cdot s_2$ and $a, b \in \Sigma$.

Consider computing a summary s for paths from node A to node E in Figure 2a. The corresponding summaries of s_1 and s_2 are $A \xrightarrow{\llparenthesis_1 \llbracket_2 \rrparenthesis_1} D$ and $F \xrightarrow{\llbracket_2 \rrparenthesis_1 \rrbracket_2} E$, respectively. Note that in LCL-reachability, path strings s_1 = "$\llparenthesis_1 \llbracket_2 \rrparenthesis_1$" and s_2 ="$\llbracket_2 \rrparenthesis_1 \rrbracket_2$" can form a valid InterDyck-word "$s = \llparenthesis_1 \llbracket_2 \rrparenthesis_1 \rrbracket_2$" with b ="$\rrbracket_2$" and a ="$\llparenthesis_1$." Therefore, the LCL-reachability algorithm obtains a new summary by combining the summaries s_1 and s_2, and the new summary indicates the node pair (A, E) is InterDyck-reachable. However, the summaries s_1 and s_2 are from two different paths, so there exists no path that realizes the InterDyck-string "$\llparenthesis_1 \llbracket_2 \rrparenthesis_1 \rrbracket_2$".

2.2 New Insights for Single-Source-Single-Target InterDyck-Reachability

For single-source-single-target InterDyck-reachability, our new algorithm considers the path expressions shown in Figure 2b. By utilizing path expressions, our algorithm can capture paths going through the same set of cycles and formulate the reachability problem as a parenthesis-counting problem. As a result, it can avoid the aforementioned sources of imprecision and conclude that node E is not InterDyck-reachable from node A in Figure 2a. Next, we discuss the insights behind path expressions and parenthesis counting.

Path expressions. Utilizing path expressions between the source node and target node can overcome the multi-path imprecision. Because our algorithm addresses a single-source-single-target reachability problem, we can indeed adopt Tarjan's method [Tarjan 1981] to compute a path expression, which represents all paths from the source to the target. In particular, a path expression is a regular expression that contains all realized strings of the paths. Consider again the example in Figure 2. The path expression between node A and node E is "$\llparenthesis_1 \llbracket_2 \rrparenthesis_1 \rrbracket_1 \mid \llparenthesis_2 (\llbracket_2 \rrparenthesis_1)^* \rrbracket_2$." As shown in Figure 2b, this path expression shows that all paths between the nodes A and E can be split into two sets: one corresponding to the expression "$\llparenthesis_1 \llbracket_2 \rrparenthesis_1 \rrbracket_1$" and the other set corresponding to the expression "$\llparenthesis_2 (\llbracket_2 \rrparenthesis_1)^* \rrbracket_2$". The first set (only a single path in this example) follows the upper path $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$. Paths in the other set go from node A to F, potentially through the cycle $F \rightarrow G \rightarrow F$ for an arbitrary number of times, and then to E. A Kleene star in a path expression represents a cycle in the path. For example, "$(\llbracket_2 \rrparenthesis_1)^*$" represents the cycle $F \rightarrow G \rightarrow F$. Thus, we can obtain the *path structures* from the path expression, *i.e.*, the cycles in the path, and how they are connected. Figure 2b gives the two path structures extracted from the corresponding path expression. Then, we can analyze every single path structure between the node pair directly. By solving the number the traversals for each cycle, we can explore every path in the path structure, thus completely addressing the multi-path imprecision.

Parenthesis-counting. The parenthesis-counting formulation can avoid approximating paths based on finite summaries. With path expressions, we can reason about paths of a fixed structure. For a specific path, we only need to fix the number of cycles traversed in the path structure. This is the key enabling insight behind formulating InterDyck-reachability as a parenthesis-counting problem based on integer linear programming (ILP). In our ILP encoding, we use a variable to represent the number of times a specific cycle is traversed. The balanced-parenthesis property in InterDyck-reachability naturally forms an ILP instance. Specifically, in Figure 2, the path-expression fragment "$\llparenthesis_1 \llbracket_2 \rrparenthesis_1 \rrbracket_1$" contains no Kleene stars; thus, we can express it as an ILP problem instance without any variables as follows:

$$\begin{aligned} 1 - 1 &= 0 \quad (\text{for } \llparenthesis_1, \rrparenthesis_1) \\ -1 &= 0 \quad (\text{for } \llbracket_1, \rrbracket_1) \\ 1 &= 0 \quad (\text{for } \llbracket_2, \rrbracket_2). \end{aligned}$$

Each equality in the ILP instance is derived from the balanced-parenthesis property. For example, the first equation "$1-1=0$" means that there exists one "$(\!|_1$" and one "$|\!)_1$" in the "$(\!|_1 [\![_2 |\!)_1]\!]_1$" expression. This ILP system is clearly infeasible. It means that all the paths following this path structure are not valid InterDyck-paths. Similarly, for the expression "$(\!|_2 ([\![_2 |\!)_1)^*]\!]_2$", we obtain another ILP system:

$$\begin{aligned} -x &= 0 \quad (\text{for } (\!|_1, |\!)_1) \\ 1 &= 0 \quad (\text{for } (\!|_2, |\!)_2) \\ x - 1 &= 0 \quad (\text{for } [\![_2,]\!]_2). \end{aligned}$$

In this ILP system, the variable x represents the number of times the cycle $F \longrightarrow G \longrightarrow F$ is traversed. This ILP system is also infeasible, which means that no paths following the corresponding path structure are valid InterDyck-paths. Unlike SPDS- and LCL-reachability, the ILP formulation for the parenthesis-counting problem does not rely on approximating paths. Consequently, we can avoid the imprecision from finite-summary approximation.

Our discussion has covered all path structures in Figure 2, and there exists no InterDyck-path between node A and node E. Thus, node E is not InterDyck-reachable from node A. Note that because InterDyck-reachability is undecidable [Reps 2000], our algorithm is inherently imprecise. One source of imprecision comes from the parenthesis-counting formulation for recognizing valid InterDyck-paths because the formulation overlooks the ordering among the parentheses.

3 PRELIMINARIES

This section gives the definitions used in our work. Section 3.1 defines Dyck languages and InterDyck languages. Section 3.2 introduces InterDyck-reachability. Section 3.3 describes path expressions and the standard form used in our algorithm.

3.1 Dyck Languages and InterDyck Languages

A Dyck language describes a set of strings with k kinds of balanced parentheses. It is a context-free language with the alphabet $\Sigma = \{(\!|_1, \cdots, (\!|_k, |\!)_1, \cdots, |\!)_k\}$ and the following grammar:

$$\begin{aligned} S &= S\ S \mid \epsilon \\ S &= (\!|_i\ S\ |\!)_i \quad \text{for } i = 1, \cdots, k. \end{aligned}$$

An interleaved Dyck (InterDyck) language, a typical non-context-free language, combines multiple Dyck languages based on an *interleaving* operator $\odot$.

Definition 3.1 (Interleaving operator). $\odot : \Sigma^* \times \Sigma^* \longrightarrow \mathcal{P}(\Sigma^*)$ is a binary operator that takes two strings and returns a set of strings, where $\mathcal{P}(\cdot)$ denotes the power-set operator. The operator $\odot$ is inductively defined as follows: for every $u \in \Sigma^*$, we have $u \odot \epsilon = \epsilon \odot u = \{u\}$. Moreover, for every $\alpha_1, \alpha_2, u_1, u_2 \in \Sigma^*$, $\alpha_1 u_1 \odot \alpha_2 u_2 = \{\alpha_1 w \mid w \in (u_1 \odot \alpha_2 u_2)\} \cup \{\alpha_2 w \mid w \in (\alpha_1 u_1 \odot u_2)\}$. The interleaving operator can be extended to languages with

$$L_1 \odot L_2 = \bigcup_{u_1 \in L_1, u_2 \in L_2} u_1 \odot u_2.$$

Note that $\odot$ is associative—*i.e.*, $(L_1 \odot L_2) \odot L_3 = L_1 \odot (L_2 \odot L_3)$—and hence can be extended to k Dyck languages with disjoint alphabets.

With the definition of the interleaving operator, we can formally describe the InterDyck language.

Definition 3.2 (InterDyck language). The InterDyck language L is the interleaving of multiple Dyck languages, *i.e.*, the language $L = L_1 \odot \cdots \odot L_k$ where k is an integer with $k \geq 2$, and each L_i is a Dyck language for every $i = 1, \cdots, k$.

3.2 InterDyck-Reachability

This work focuses on InterDyck-reachability, which belongs to the general formal-language-reachability framework [Reps 1998].

Definition 3.3 (Formal-language-reachability problem). Given a formal language L and a directed edge-labeled graph $G = (V, E)$ with each edge $e \in E$ labeled by a terminal $t \in \Sigma$, we say that a path $p = e_1 e_2 \ldots e_m$ in G *realizes* a string $R(p)$ over the alphabet Σ by concatenating the edge labels in the path in order, *i.e.*, $R(p) = t_1 t_2 t_3 \ldots t_m$ where t_i is the label for e_i. A path p in G is an *L-path* if the realized string $R(p)$ is a word in the formal language L. Node v is *L-reachable* from node u iff there exists an L-path from u to v in G. The formal-language-reachability problem for L, abbreviated as the *L-reachability* problem $Reach\langle L, G\rangle$, is to compute all the L-reachable node pairs in the graph G. The single-source-single-target variant of the L-reachability problem for a node pair (u, v) is to compute whether v is L-reachable from u.

Using this definition, we now define the InterDyck-reachability problem.

Definition 3.4 (InterDyck-reachability). The InterDyck-reachability problem is a formal-language-reachability problem $Reach\langle L, G\rangle$ with L being an InterDyck language (Definition 3.2). The single-source-single-target variant of the problem is to decide whether a given node pair u, v is L-reachable.

3.3 Path Expressions and Integer Linear Programming (ILP) Standard Form

Our work solves the single-source-single-target reachability problem by generating *path expressions* and then formulating InterDyck-reachability as an InterDyck-path recognition problem via parenthesis counting. We formally define path expressions as follows.

Definition 3.5 (Path expressions). Given an edge-labeled graph $G = (V, E)$, the path expression from node u to node v is a regular expression RegExp satisfying the following conditions: (1) For every path p between nodes u and v, its realized string $R(p)$ is a word in the regular language defined by RegExp; and (2) conversely, for any word $w \in$ RexExp, there exists a path p from u to v such that its realized string $R(p) = w$.

Based on parenthesis counting, our reachability algorithm requires the path expressions to be in an integer linear programming (ILP) standard form.

Definition 3.6 (ILP standard form). A path expression RegExp is in the ILP standard form if it has the form of RegExp $= t_1 | t_2 | \ldots | t_N$, where each t_i for $i = 1 \ldots . N$ is a regular expression without any occurrences of the "or" operation.

Any regular expression R can be transformed into a regular expression R_{ILP} in the ILP standard form that generates the same language (*i.e.*, $L(R) = L(R_{ILP})$)—see Section 4.2

4 APPROACH

4.1 Overview

Our algorithm takes as input a triple (G, s, t), where G is an edge-labeled graph and s and t are the source and target nodes, respectively. The algorithm requires the input graph G to be a finite graph and to be fully available at the time the query is issued. We specifically do *not* address the situation in which we are given s and t, and the algorithm has the ability to explore some other structure—such as an abstract-syntax tree—from which G can be constructed on the fly.

Algorithm 1 gives our single-source-single-target InterDyck-reachability algorithm. The algorithm first applies Tarjan's method [Tarjan 1981] to generate a path expression in the ILP standard form on line 1. Section 4.2 discusses the path-expression generation. For each "disjunct" (*i.e.*, each

Algorithm 1: Main algorithm for single-source-single-target InterDyck-reachability.

```
Input  : Edge-labeled graph $G = (V, E)$, a start node $s$, and a target node $t$;
Output : A Boolean indicating whether node $t$ is InterDyck-reachable from node $s$ in $G$.
1 RegExp ← TarjansMethod ($G, s, t$) ;                      // RegExp is in ILP standard form
2 foreach sub regular expression $t_i$ of path expression RegExp = $t_1 | \dots | t_N$ do
3     LP_instance ← GenerateLPInstance ($t_i$)
4     if SolveLP(LP_instance) is True then
5         return True
6 return False
```

or-operator-separated sub-expression), the algorithm generates a set of constraints to obtain an ILP instance (line 3). If an ILP solver concludes that the constraints are solvable, our algorithm returns that the target node t is InterDyck-reachable from the source node s (line 5).

Section 4.3 discusses how to encode path expressions as ILP instances. Note that the ILP standard form can contain an exponential number of disjuncts in the worst case. Section 4.4 further presents a technique to reduce the number of terms in the ILP form. Our experiments (Section 5.4) indicate that the worst-case exponential blow-up does not occur in practice.

4.2 Generating Path Expressions

Path expressions are central to our InterDyck-reachability algorithm, *i.e.*, our algorithm decides reachability between nodes s and t based on the corresponding ILP instances obtained from a path expression from node s to node t.

Tarjan's method. We use Tarjan's method [Tarjan 1981] to generate a regular expression RegExp. The path expression is a succinct representation of the realized strings of all the paths between s and t. Tarjan's method can generate path expressions in $O(|E|\alpha(|E|))$ time, where $|E|$ is the number of edges in the graph and α is the inverse Ackermann function. It assumes that graph G is fully known before constructing the path expression. Consequently, our algorithm does not apply to a situation in which a graph is being explored on the fly, and the goal is to test InterDyck-reachability along newly discovered paths (and possibly sped up by refuting partial paths).

Regular-expression representation. The choice of the representation of path expressions plays a key role in our algorithm. We use a hash-consed [Goto 1974] acyclic-graph representation (HCAG) of regular expressions. The HCAG representation consists of four types of nodes: terminal nodes, concatenation nodes, *or* nodes, and star nodes. Each terminal in the regular expression corresponds to a terminal node in the graph. The concatenation nodes and *or* nodes are binary nodes. A concatenation node with two children a and b (ordered left-to-right) represents the regular expression "$a \cdot b$." Similarly, an *or* node with two children a and b represents the regular expression "$a \mid b$." A star node is a unary node; if the child node represents a regular expression a, the star node corresponds to the Kleene star "a^*." In the HCAG representation, different nodes represent different regular expressions; each regular expression also has a unique representative node in the HCAG.

Example 4.1 (HCAG representation). Consider the regular expression "$(\llparenthesis_1^* \llbracket_1 | \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*$." Figure 3 depicts its HCAG representation. Each node in the graph represents a unique regular sub-expression. For example, the sub-expression "$\llparenthesis_1^*$" appears twice in the regular expression. But there is only one corresponding node in the graph, which is the leftmost star node. In this example, its two in-edges represent there are two occurrences of this expression in the regular expression: it serves as the operand for two different concatenation operations.

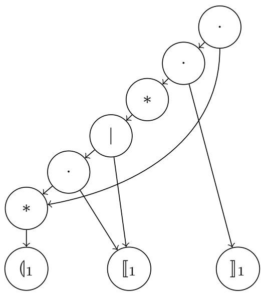


Fig. 3. HCAG representation of the regular expression “$(\llparenthesis_1^* \llbracket_1 | \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*$.”

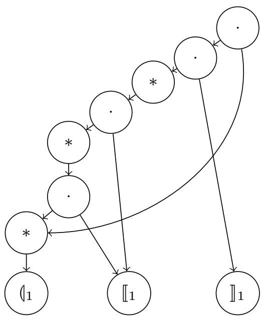


Fig. 4. HCAG representation of the ILP standard form “$((\llparenthesis_1^* \llbracket_1)^* \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*$” for the regular expression “$(\llparenthesis_1^* \llbracket_1 | \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*$” in Figure 3.

During path-expression generation, we normalize regular expressions into the ILP standard form. Indeed, we can obtain the ILP standard form of a regular expression using the transformations:

- $(a|b)^* = (a^* b)^*$
- $(a|b) \cdot c = (ac|bc)$.

Because the above rules cover both the “$*$” and “$\cdot$” operations, it is then immediate that we can always generate an ILP standard form for any regular expression.

Theorem 4.2. *Every regular expression has an ILP standard form.*

Example 4.3 (ILP standard form). Consider the regular expression “$(\llparenthesis_1^* \llbracket_1 | \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*$” in Example 4.1. By applying the transformation rules, we can obtain a regular expression in the ILP standard form: $((\llparenthesis_1^* \llbracket_1)^* \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*$. Figure 4 depicts the HCAG representation of this regular expression.

4.3 ILP Encoding

Algorithm 2 encodes the path expression as an ILP instance. The top-level sub-expressions of a path expression in ILP standard form—its disjuncts—provide a kind of global view of the paths between s and t. The Kleene-star operators in a disjunct represent the cycles in the paths. The idea behind our ILP encoding is to assign variables to the cycles in the path. Each variable represents the number of times the path goes through the associated cycle.

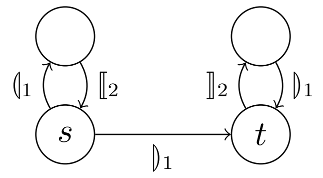


Fig. 5. Path structure for the regular expression "$((\!|_1 [\![_2)^* |\!)_1 (]\!]_2 |\!)_1)^*$."

Consider a regular expression "$((\!|_1 [\![_2)^* |\!)_1 (]\!]_2 |\!)_1)^*$." This regular expression corresponds to the path structure in Figure 5 with two cycles, C_1 and C_2. The realized strings $R(C_1)$ of the left cycle C_1 and $R(C_2)$ of the right cycle C_2 are "$(\!|_1 [\![_2$" and "$]\!]_2 |\!)_1$," respectively. A path from s to t could go through each of C_1 and C_2 arbitrarily many times. We associate a variable x_1 with the cycle C_1, and a different variable x_2 with the cycle C_2. For this path expression, the generated ILP constraints are:

$$\begin{aligned} x_i &\geq 0 \\ x_1 - 1 - x_2 &= 0 \quad (\text{for } (\!|_1, |\!)_1) \\ x_1 - x_2 &= 0 \quad (\text{for } [\![_2,]\!]_2). \end{aligned}$$

The first inequality $x_i \geq 0$ is due to the fact that the path cannot traverse a cycle for a negative number of times. The second equation comes from the balanced parenthesis property for "$(\!|_1$". Similarly, we can obtain the last equation for "$[\![_2$". After generating the ILP instance, we utilize existing ILP solvers to obtain a solution. If the ILP instance is feasible, we conclude that the node t is InterDyck-reachable from s. In this example, the ILP problem is infeasible because $x_1 - 1 - x_2 = 0$ contradicts with $x_1 - x_2 = 0$. The contradiction indicates that the path expression $((\!|_1 [\![_2)^* |\!)_1 (]\!]_2 |\!)_1)^*$ does not admit any valid InterDyck-paths.

ILP-constraint generation. We generate ILP constraints in a bottom-up traversal of the HACG representation. Algorithm 2 employs a three-dimensional table "count" for ILP-constraint generation. For each regular-expression node v, there exists a two-dimensional slice of the count table denoted as count$[v]$. The entry count$[v][i][j]$ of the slice contains a template to generate linear expressions that represent the number of unmatched open parentheses for the j^{th} parenthesis symbol of the i^{th} Dyck-language. For example, for the regular expression $((\!|_1 [\![_1)^* ((\!|_1 (\!|_1)^*$, the count table contains the template "$x + 2 \cdot y$" in the entry count$[v][1][1]$ (if we regard parentheses as the first Dyck language and brackets are the second) and the template "x" in the entry count$[v][2][1]$. Within the count table for the same node, if there exist occurrences of identical variables for different parentheses/brackets, they refer to the same variable; *i.e.*, in this example, the variable x in count$[v][1][1]$ and count$[v][2][1]$ can be shared. Notice that even though "$(\!|_1$" and "$[\![_1$" are in different Dyck languages, when their templates share a variable, it means they are in the same cycle. Therefore, we can re-use the common variable. However, this does not apply to the case where one variable in the count table corresponds to two different nodes in the graph. Intuitively, for two different nodes, there is no correspondence between the number of times each appears in the paths. For instance, if there are two templates "$x + y$" and "$2 \cdot x + y$" in the count table for two different nodes, the "sum" of their templates is "$x_1 + y_1 + 2 \cdot x_2 + y_2$" instead of "$3 \cdot x + 2 \cdot y$."

Algorithm 2: GenerateLPInstance(t) – ILP constraint generation.

```
Input  : A path expression $t$ without "or" nodes, represented by acyclic graph $G = (V, E)$;
Output : A constraint table $T$ for each node.
1  Topologically sort the node set $V$ in ascending order in $G$, i.e., the ordering of a node is always larger than
   its children nodes.
2  foreach node $v \in V$ do
3      if node $v$ represents a terminal then
4          if the terminal is $(\!|_j$ in the $i^{th}$ Dyck language then
5              count[$v$][$i$][$j$] = 1
6          if the terminal is $|\!)_j$ in the $i^{th}$ Dyck language then
7              count[$v$][$i$][$j$] = -1
8      if node $v$ represents a concatenation operator then
9          Let $v_1, v_2$ be the two children nodes of $v$
10         if bad[$v_1$] is True or bad[$v_2$] is True then
11             bad[$v$] =True; continue
12         count[$v$] = count[$v_1$] + count[$v_2$]
13     if node $v$ represents a star operator then
14         Let $v_1$ be the child node of $v$
15         if bad[$v_1$] is True then
16             continue
17         let $x$ be a fresh variable
18         foreach $(i, j)$ with $j$ representing the parenthesis index in the $i^{th}$ Dyck language do
19             if count[$v_1$][$i$][$j$] contains any variable then
20                 let $y$ be a fresh variable
21                 count[$v$][$i$][$j$] = $y$
22             else
23                 count[$v$][$i$][$j$] = count[$v_1$][$i$][$j$] $\cdot x$
24 return count
```

To generate the count table, during the bottom-up traversal, when encountering a star operator, the algorithm allocates a variable associated with the cycle. When encountering a concatenation node, we generate a new count table entry based on the sum of its child nodes' entries. Therefore, the count table can encode the property that the number of open parentheses should be equal to the number of close parenthese in any valid InterDyck-paths. We now proceed to discuss Algorithm 2.

- **Line 1**: We first sort nodes in topological order. The topological sort ensures that the ordering of a parent node is always greater than the ordering of the children nodes. Then, the algorithm performs a bottom-up graph traversal by filling the count table in ascending order.
- **Lines 3-7**: We consider three types of nodes: (1) terminal nodes, (2) concatenation nodes, and (3) star nodes. Note that Algorithm 2 does not need to consider the "or" operator because "or" operators have been lifted to the top-level in the ILP standard form. Each disjunct of a regular expression in the ILP standard form is considered separately by Algorithm 2. Lines 3-7 handle terminal nodes. The count table maintains the number of unmatched open parentheses for each type of parenthesis symbol. The algorithm set the corresponding count entry value to "1" if the terminal is an open parenthesis. Otherwise, it sets the entry to "−1". The values in

entries of count table can be negative, which indicates that the corresponding expressions contain unmatched close parentheses.

- **Lines 8-12**: For a concatenation node, we obtain the number of unmatched parentheses by computing the sum of its two children nodes. Specifically, line 12 assigns the summation result to the count-table entry. On lines 10-11, when the sub-expression cannot be part of a valid InterDyck word, it skips the count-table computation. We associate each node with a Boolean variable bad (line 10). If the bad value is true, its corresponding sub-expression cannot be involved in any valid InterDyck words. In Section 4.4 we have a further discussion on the computation of the bad variable.
- **Lines 13-23**: Each Kleene star represents a potential cycle in the path. On line 17, the algorithm first obtains a new variable representing the number of times the cycle is traversed in the path. On line 23, each unmatched parenthesis is multiplied by the new variable. Note that when the count table entry already contains a variable, we cannot simply multiply it with the new variable because it would otherwise make the result non-linear. Instead, we replace the result with another new variable (lines 19-21). Maintaining linearity is key to over-approximating the balanced-parenthesis property with parenthesis counting.

With the count table, for each regular sub-expression t, we generate ILP constraints to guarantee that the number of open parentheses is equal to the number of close parentheses. Specifically, for every entry in $\mathrm{count}[t]$, we add a new constraint $\mathrm{count}[t][i][j] = 0$. If the application allows unmatched open parentheses, we can also set the constraint to be $\mathrm{count}[t][i][j] \geq 0$. In addition, if the entry $\mathrm{bad}[t]$ is True, we can skip constraint generation and directly report that the ILP instance is not solvable. Therefore, we obtain an ILP problem instance (line 3 in Algorithm 1). The next step is to solve the ILP problem instance with an ILP solver. If the problem instance is feasible, the algorithm reports that t is InterDyck-reachable from s.

Example 4.4 (ILP constraint generation). Consider the regular expression $((\llparenthesis_1^* \llbracket_1)^* \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*$. Figure 4 shows the HCAG representation of the regular expression. Algorithm 2 traverses the nodes in a bottom-up manner. It first generates the count table for the node representing the terminal "$\llparenthesis_1$" (lines 3-7). Because "$\llparenthesis_1$" is an open parenthesis, line 5 sets $\mathrm{count}[\llparenthesis_1][1][1]$ to 1. The algorithm then considers the node that represents "$\llparenthesis_1^*$". It is a star-operator node, which is handled by the branch on lines 13-23. Because the count table of its child node "$\llparenthesis_1$" already contains $\mathrm{count}[\llparenthesis_1][1][1] = 1$, the algorithm sets $\mathrm{count}[\llparenthesis_1^*][1][1]$ to $1 \cdot x = x$ on line 23. Similarly, the algorithm iterates over the remaining nodes of the regular expression to complete the count table. Finally, we have $\mathrm{count}[((\llparenthesis_1^* \llbracket_1)^* \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*][1][1] = y_1 + x_1$ and $\mathrm{count}[((\llparenthesis_1^* \llbracket_1)^* \llbracket_1)^* \rrbracket_1 \llparenthesis_1^*][2][1] = y_2 - 1$. As a result, the constraint for "$\llparenthesis_1$" is $y_1 + x_1 = 0$ and the constraint for "$\llbracket_1$" is $y_2 - 1 = 0$.

Over-approximation in ILP constraint generation. Because the InterDyck-reachability problem is undecidable, and the ILP problem is NP-complete, the ILP constraints indeed over-approximate InterDyck-reachability. On lines 19-21, we over-approximate a non-linear expression using a new variable, which does not rely on existing variables. The second source of over-approximation is that for the InterDyck language, the ordering among different types of parentheses in the same Dyck language is not strictly enforced in the ILP constraints. For example, for the regular expression $\llparenthesis_1^* \llparenthesis_2 \rrparenthesis_1 \llbracket_2$ as a prefix for InterDyck-word, the non-negativity prefix constraints generated for parenthesis "$\llparenthesis_1$" ($x - 1 \geq 0$) and parenthesis "$\llparenthesis_2$" ($1 \geq 0$) cannot express the property that the close parenthesis "$\rrparenthesis_1$" cannot immediately follow the open parenthesis "$\llparenthesis_2$" in any valid Dyck strings. Section 4.4 discusses a technique to mitigate this type of over-approximation by utilizing the bad variable in Algorithm 2.

Algorithm correctness. Because Tarjan's method [Tarjan 1981] always computes correct path expressions, our correctness discussion focuses on demonstrating that the ILP instances obtained from the path expressions over-approximate InterDyck-reachability. In Algorithm 2, the count table maintains the number of unmatched open parenthesis for every sub-expression in the path expression. Each ILP instance guarantees that the number of open parentheses is the same as the close parentheses along any valid InterDyck-path; thus the unmatched open parentheses should always be zero. This constraint is a necessary condition for the path to be an InterDyck-path. Thus, the ILP solution over-approximates the InterDyck-reachability solution. The correctness of our proposed single-source-single-target reachability algorithm follows.

4.4 Improving Parentheses Counting

If we compute InterDyck-reachability only based on ILP instances, Algorithm 2 needs to generate an ILP instance for each sub-expression t_i in the ILP standard form "$t_1 | t_2 | \ldots | t_N$." Theoretically, the number N can be exponentially large. Thus the approach can potentially be time-consuming. Moreover, the parenthesis-counting formulation cannot capture parenthesis ordering in valid Dyck- and InterDyck-paths. For example, consider a valid Dyck string "$⦇_1 ⦇_2 ⦈_2 ⦈_1$" in the parenthesis Dyck language. Parenthesis counting cannot express the constraint that to cancel the unmatched "$⦇_1 ⦇_2$", the close parenthesis "$⦈_2$" must appear before "$⦈_1$." Our algorithm integrates LCL-reachability to address the aforementioned shortcomings. In particular, we utilize LCL-reachability to avoid considering sub-expressions that do not contain any valid InterDyck words and reduce the number of terms in the ILP standard form. Therefore, it improves both the precision and performance of the parenthesis counting formulation.

Example 4.5 (Improving precision via LCL-reachability). Consider the path with the realized string "$⦇_1 ⟦_1 ⦇_2 ⦈_1 ⦈_2 ⟧_1$." The parenthesis-counting formulation cannot recognize this invalid InterDyck-path because it does not consider the parenthesis ordering. Specifically, the close parenthesis "$⦈_1$" should not immediately follow the open parenthesis "$⦇_2$." However, by integrating LCL-reachability, we can eliminate this invalid path. The LCL-reachability algorithm can reject this string because the substring "$⦇_2 ⦈_1$" cannot be expressed by any valid LCL grammar rules. Therefore, LCL-reachability captures some parenthesis-ordering restrictions and improves the precision of our algorithm.

However, LCL-reachability, being a graph algorithm, cannot directly work on path expressions. To incorporate LCL reachability, our algorithm builds the *derived graphs*. Specifically, given a path-expression node that represents the regular expression e in HCAG, its derived graph is the minimal graph with two distinct nodes s and t, and the path expression between them is e. In other words, the nondeterministic finite automaton (NFA) diagram of the path expression e can be regarded as the derived graph for e.

Example 4.6 (Derived graph). Figure 5 gives the derived graph for the regular expression "$((⦇_1 ⟦_2)^* ⦈_1 (⟧_2 ⦈_1)^*$," which is similar to the corresponding NFA.

The LCL-reachability algorithm computes reachability summaries between any nodes in the derived graphs (including s and t). Every sub-expression in the HCAG corresponds to a node pair in the derived graph. We then propagate the corresponding LCL-reachability summaries based on a bottom-up graph traversal of the HCAG. Therefore, the LCL-reachability algorithm can use the summaries to decide whether a particular sub-expression can belong to an InterDyck-word. If not, the corresponding flag $\mathrm{bad}(v)$ becomes *True*, and Algorithm 2 can avoid generating ILP instances for such infeasible sub-expressions.

5 EVALUATION

This section evaluates our single-source-single-target InterDyck-reachability algorithm based on two client analyses. We compare the algorithm against two state-of-the-art InterDyck-reachability algorithms, LCL-reachability [Zhang and Su 2017] and SPDS-reachability [Späth et al. 2019]. In addition, we also compare the algorithm with two domain-specific analysis tools. The evaluation focuses on addressing three research questions.

- **RQ1**: How does our algorithm compare to the state-of-the-art InterDyck-reachability algorithms, in terms of performance, precision, and memory consumption?
- **RQ2**: How does our algorithm compare to domain-specific analysis tools?
- **RQ3**: Does our algorithm exhibit any exponential behavior in practice?

5.1 Experimental Setup

Benchmarks. Our evaluation is based on two suites of benchmarks.

- *DDVerify benchmark.* DDVerify [Witkowski et al. 2007] is a concurrent program state-reachability analysis benchmark consisting of a set of concurrent Boolean programs with assertions. The context-sensitive formulation of the program state reachability can be transformed into an InterDyck-reachability problem on their state-transition graphs. The starting state of the Boolean program and the state that violates an assertion can be translated as the (source, target) node pair for single-source-single-target InterDyck-reachability.
- *Demand-driven taint-analysis benchmark.* The second client analysis adopts the taint-analysis benchmark of Huang et al. [2015]. The benchmark consists of a set of Android applications obtained from the Google Play store. For each application, there is a query between a specified source and a specified sink for the single-source-single-target InterDyck-reachability.

Graph generation. We extract edge-labeled graphs from the two benchmark suites.

- The DDVerify benchmark consists of Boolean programs obtained via predicate abstraction. DDVerify supports programs that use run-time thread creation. Because our algorithm only works on finite graphs, we selected 30 Boolean programs with a fixed number of threads and translated them into graphs for InterDyck-reachability. Nodes in the graphs represent program states. For example, consider a program with one Boolean variable x and two threads. Each node denotes a tuple (v_x, t_1, t_2) where v_x represents the value of the Boolean variable x, t_1 represents the line to be executed in the first thread, and t_2 represents the line to be executed in the second thread. Each edge represents a program state transition $(v_x, t_1, t_2) \longrightarrow (v'_x, t'_1, t'_2)$ after one step of execution. If the transition denotes a function call/return, the edge is labeled by an open/close parenthesis from the Dyck language associated with the executing thread. Therefore, for a program with t threads, the corresponding InterDyck language represents the interleaving of t Dyck languages. Finally, the assertion check becomes a single-source-single-target InterDyck-reachability query on the translated graphs.
- For the taint-analysis benchmark, we used DroidInfer [Huang et al. 2015] to produce the interprocedural control-flow graph for each benchmark program. Nodes represent variables in the programs, and edges represent value flows between them. We labeled the edges related to function calls and returns with the open-parenthesis and close-parenthesis labels from one Dyck language. We used a second Dyck language to label edges related to field writes and reads. Therefore, each taint-analysis query becomes an instance of the single-source-single-target InterDyck-reachability problem.

Graph simplification. Our work focuses on InterDyck-reachability. Therefore, the graph-simplification technique described in Li et al. [2020] is directly applicable. In our experiments, we compare the InterDyck-reachability algorithms on both original graphs and simplified graphs.

Implementation. We implemented all algorithms in C++.[1] The executables were compiled using g++ with the "-O2" optimization flag. We used the Gurobi Optimizer [Gurobi Optimization, LLC 2021] as the backend ILP solver. All experiments were conducted on a server with two AMD EPYC 7402 CPUs and 512GB RAM, running Ubuntu 20.04.

Evaluated approaches. We compare our algorithm against the state-of-the-art *all-pairs* InterDyck-reachability algorithms and the domain-specific analysis tools for the two benchmarks. The evaluated InterDyck-reachability algorithms include LCL-reachability [Zhang and Su 2017] and SPDS-reachability [Späth et al. 2019]. Note that the LCL-reachability algorithm only works for the all-pairs case. For SPDS-reachability, the original demand-driven implementation is tied to the underlying analysis infrastructure. Therefore, we re-implemented the all-pairs version for both. For the domain-specific analysis, we applied CUBA [Liu and Wahl 2018] and DroidInfer [Huang et al. 2015] to the two benchmarks, respectively. Specifically, CUBA is a static verifier for concurrent recursive programs based on interprocedural context-unbounded analysis. It implements a semi-decision procedure in which the number of context switches is considered to be a resource parameter. It means that CUBA may never terminate unless an explicit context-switch bound is provided. DroidInfer is a context-sensitive taint-analysis tool to detect privacy leaks in Android applications. The implementations of the two domain-specific tools are publicly available.[2]

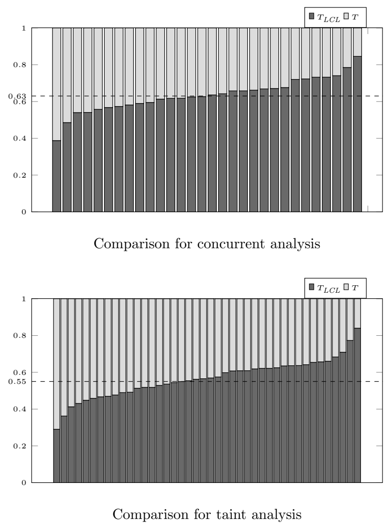
5.2 RQ1: Comparisons with InterDyck-Reachability Algorithms

We first compare our single-source-single-target algorithm against two state-of-the-art all-pairs InterDyck-reachability algorithms: LCL-reachability and SPDS-reachability. The experiments are based on two sets of graphs: the original graphs obtained from the benchmarks and the graphs after applying the graph simplification technique [Li et al. 2020]. The evaluation focuses on the precision, performance, and memory consumption differences among the algorithms.

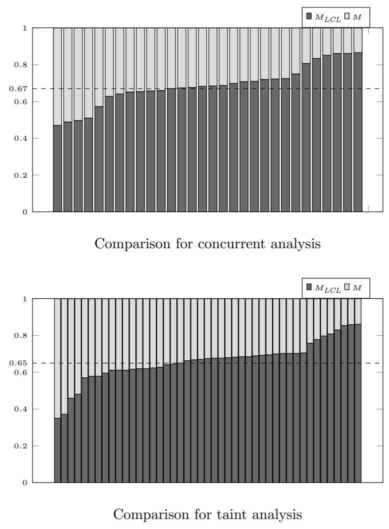
Comparison with LCL-reachability. Figure 6 portrays the results for running time and memory consumption comparisons. In particular, Figure 6a presents a bar chart that compares the running time of our algorithm and the LCL-reachability algorithm on the original graphs. The x-axis in Figure 6a represents the benchmark graphs; the y-axis represents the running time comparison of LCL-reachability (T_{LCL}) and our algorithm (T). The ratio of T_{LCL} and T is proportional to the ratio of the lengths of the bars. Thus, when the color changes at 0.5, the two algorithms take an equal amount of time. The benchmarks are sorted according to the length of the dark bars, with the shortest on the left. In the experiment, we set a ten-minute budget for each graph. If one algorithm does not finish within the time limit, we regard the running time as ten minutes. We also exclude cases that finish in less than 0.01 seconds. For both benchmarks, our algorithm runs faster than the all-pairs LCL-reachability algorithm—the light area is smaller than the dark area. In terms of geometric means, the running time of our algorithm is 1.42× and 1.20× faster than the all-pairs LCL-reachability algorithms on the concurrency and taint-analysis benchmarks, respectively.

Figure 6b presents a bar chart that compares the memory usage between LCL-reachability and our algorithm. The x-axis represents the benchmark graphs, and the y-axis represents the comparison between LCL-reachability M_{LCL} and our algorithm M. The ratio of M_{LCL} and M is

[1]Our implementation is publicly available at https://doi.org/10.5281/zenodo.7299969.
[2]The implementation of CUBA can be found at https://github.com/lpzun/cuba. The implementation of DroidInfer can be found at https://github.com/proganalysis/type-inference.



(a) Comparison of running times between our algorithm and LCL-reachability.



(b) Comparison of memory consumption between our algorithm and LCL-reachability.

Fig. 6. Comparisons with LCL-reachability. In each bar chart, the dotted line represents the geometric mean of the bar length ratios.

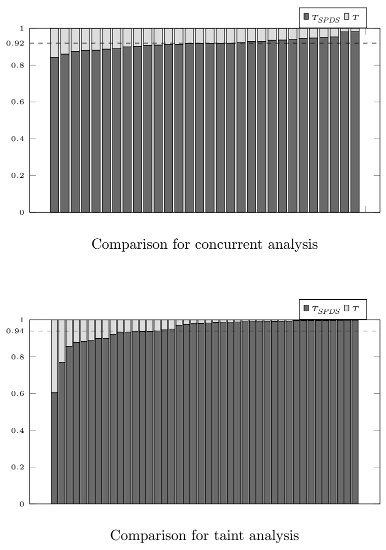
proportional to the ratio of the lengths of the bars. If the ratio is at 0.5, the two algorithms take an equal amount of memory. Our algorithm consumes less memory compared with the all-pairs LCL-reachability algorithm because the area of the dark bars is larger than that of the light bars. In terms of geometric means, our algorithm consumes 57% and 53% less memory than LCL-reachability on the concurrency and taint-analysis benchmarks, respectively.

For precision comparison, our algorithm can recognize the same number of reachable node pairs as in the concurrent program state-reachability analysis. For demand-driven taint analysis, our algorithm reports 4.55% fewer reachable node pairs. Because both algorithms are over-approximation algorithms, the 4.55% of node pairs excluded by our algorithm are all false positives.

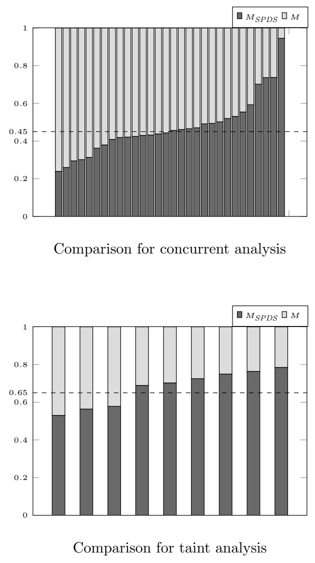
Comparison with SPDS-reachability. Figure 7 presents the comparison results. In particular, Figure 7a demonstrates the running time comparison. As before, the x-axis represents the benchmark graphs; the y-axis shows the comparison of our running time T and the running time T_{SPDS} of SPDS-reachability. The experiments demonstrate that our algorithm has substantially better running time than the SPDS-reachability algorithm because the light area in the bar chart is significantly smaller than the dark area. For geometric means, our algorithm exhibits 28.24× and 69.9× speedup compared with SPDS-reachability on the concurrency and taint-analysis benchmarks, respectively.

Figure 7b demonstrates the memory consumption comparison. Because all runs are performed under a time budget, the bar charts only show results for runs in which both algorithms finished within the time budget. In terms of geometric means, our algorithm consumes 10.4% more memory and 59.4% less memory on the concurrency and taint-analysis benchmarks, respectively.

Our algorithm also achieves better precision results. Because both the SPDS-reachability and our algorithm are over-approximation algorithms, an algorithm is more precise if it identifies fewer



(a) Comparison of running times between our algorithm and SPDS-reachability.



(b) Comparison of memory consumption.

Fig. 7. Comparisons with SPDS-reachability.

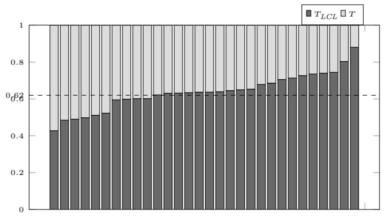
InterDyck-reachable node pairs. In the experiments, our algorithm reports 6.7% and 11.1% fewer reachable node pairs in the concurrency and taint-analysis benchmarks, respectively.

Comparison over simplified graphs. We also perform the comparisons based on simplified graphs produced by Li et al. [2020]. Figure 8 presents the performance-comparison results. Again, we use the x-axis to represent different benchmark graphs, and the y-axis to illustrate the performance comparison. The bar charts show that our algorithm achieves a slightly better result than the all-pairs LCL-reachability algorithm and significantly better running time than the SPDS-reachability algorithm. In terms of the geometric mean, the speedups compared to LCL-reachability are 1.21× and 1.11× for the two benchmarks, and the speedups compared to SPDS-reachability in two benchmarks are 22.19× and 52.9×, respectively.

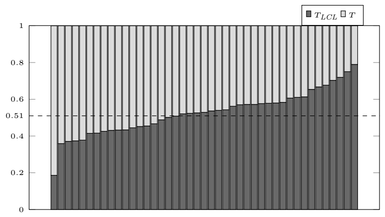
5.3 RQ2: Comparisons with Domain-Specific Analysis Tools

Besides the state-of-the-art InterDyck-reachability algorithms, we compare our algorithm with the corresponding domain-specific analysis tools, CUBA [Liu and Wahl 2018] and DroidInfer [Huang et al. 2015], for the two benchmarks. We report the results using geometric means.

Concurrent-program state-reachability analysis. For concurrent-program state-reachability analysis, Figure 9a shows the running-time comparison between CUBA and our algorithm. The area of the light bars is much smaller than that of the dark bars. It indicates the running time of our algorithm is much better than that of CUBA. On average, computed as the geometric mean, our algorithm exhibits an 11.76× speedup compared to CUBA. For precision, it is not possible to conduct an apples-to-apples comparison between CUBA and our algorithm because CUBA is a semi-decision procedure that reports an under-approximation of the true answer (which can be

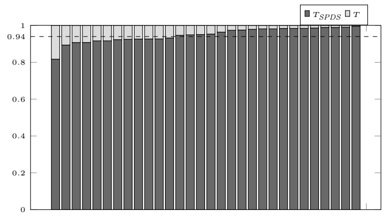


Comparison for concurrent analysis

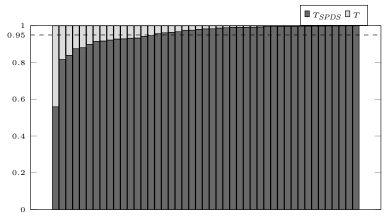


Comparison for taint analysis

(a) Comparison of running times between our algorithm and LCL-reachability on simplified graphs.



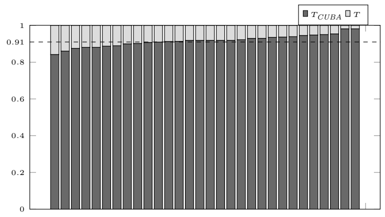
Comparison for concurrent analysis



Comparison for taint analysis

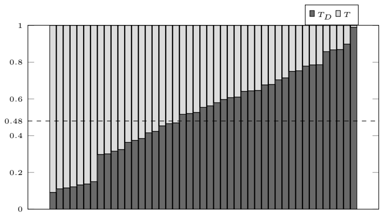
(b) Comparison of running times between our algorithm and SPDS-reachability on simplified graphs.

Fig. 8. Running-time comparisons against InterDyck-reachability algorithms on simplified graphs.



Comparison for concurrent analysis

(a) Comparison of running times between our algorithm and CUBA.



Comparison for taint analysis

(b) Comparison of running times between our algorithm and DroidInfer.

Fig. 9. Comparisons with domain-specific tools.

made more precise by increasing the value of a certain resource parameter), whereas our algorithm reports an over-approximation of the true answer.

Even though our algorithm and CUBA are not comparable in terms of precision, the results for the number of reachable queries reported by CUBA can help to understand the precision of our algorithm. In particular, for the concurrency benchmark, the number of reachable node-pair queries reported by CUBA is around 73% of what our algorithm reports. Thus, an upper bound on our algorithm's false-positive rate is 37%, whereas an upper bound on the SPDS false-positive rate is 46.1%. The CUBA precision result also provides insights into interpreting the 6.7% improvement with respect to SPDS. Our proposed algorithm removes at least 19.7% of the false-positive results.

Demand-driven taint analysis. Figure 9b presents a bar chart that compares the running times of DroidInfer and our algorithm. The bar chart does not show any clear performance advantage for either DroidInfer or our algorithm. On geometric means, our algorithm runs around 2% slower than DroidInfer. As for precision, both the DroidInfer and our algorithm are over-approximation algorithms. Our algorithm reports 6.8% fewer reachable node pair queries than DroidInfer. It indicates that the proposed algorithm is more precise.

5.4 RQ3: Exponential Blow-Up in Practice

In Algorithm 2, the number of variables contributed by a single expression node to the ILP constraints can be exponential in the size of the regular expression. This worst-case exponential complexity is a concern for the performance of the algorithm. In the experiments, we record the number of variables generated for the ILP constraints for each regular-expression node (where the regular expression is represented in ILP standard form).

Given a regular-expression node t, let N_{t_1} be the number of variables from the node in the ILP constraints and N_{t_2} be the number of variables from its children nodes in the ILP constraints. We define the number of variables generated for the node t to be $|N_{t_1} - N_{t_2}|$. If the algorithm suffers from an exponential blow-up, we expect the number $|N_{t_1} - N_{t_2}|$ to be large. For each benchmark, we compute the following two metrics to measure the blow-up of the generated ILP constraints. The metrics are (i) the maximal, and (ii) the average number of variables generated for an expression node. The maximal number of variables generated across all benchmark programs provides a worst-case measure. The average number of variables provides an average-case measure.

For the concurrency benchmark, the maximal number of variables generated for regular-expression nodes ranges from 4 to 29. Similarly, for the taint analysis, the maximal number of variables generated for regular-expression nodes ranges from 3 to 24. On average, the numbers of variables generated for regular-expression nodes are 4.3 and 7.5 for the two benchmarks, respectively. These two numbers show that there does not exist an exponential blow-up in the size of the generated ILP constraints. As a result, our algorithm does not appear to exhibit an exponential running time in practice.

Summary. Finally, we summarize our experimental findings as follows.

- *Comparisons with state-of-the-art InterDyck-reachability algorithm.* (1) Running time: our algorithm exhibits 1.20× and 1.42× speedup compared with all-pairs LCL-reachability on the evaluated benchmarks, respectively. It also has achieved 28.24× and 69.9× speedup compared with all-pairs SPDS-reachability. (2) Precision: our algorithm has achieved better precision results compared with the LCL- and SPDS-reachability algorithms, reducing the number of false-positive node pairs by up to 11.1%. (3) Memory consumption: our algorithm consumes less than 50% of the memory used by the all-pairs LCL-reachability algorithm. Compared with the SPDS-reachability algorithm, our algorithm consumes 59.4% less memory for the taint-analysis benchmark and uses a similar amount of memory for the concurrency benchmark. (4) On the simplified graphs produced by Li et al. [2020], our algorithm exhibits performance improvements similar to the original graphs.
- *Comparisons with domain-specific analysis tools.* (1) Running time: the experiments show that our algorithm is more than 10× faster than CUBA, and about comparable in speed with DroidInfer. (2) Precision: it is not possible to conduct an apples-to-apples comparison

of precision between CUBA and our algorithm. Our algorithm achieves a 6.8% precision improvement over DroidInfer on the taint-analysis benchmark.

- *Exponential worst-case growth.* In our experiments, the average numbers of variables generated from individual regular-expression nodes are 4.3 and 7.5 for the two benchmark suites. The maximum numbers of variables across all programs in the two benchmark suites are 29 and 24 for concurrency analysis and taint analysis, respectively. These numbers show that even though the theoretical worst case is exponential growth, the number of variables is small in practice.

6 RELATED WORK

6.1 Language Reachability for Static Analysis

Formal-language reachability is one of the most popular formulations for static-analysis applications. Context-free language (CFL) reachability is one of the oldest formulations. Yannakakis [1990] proposed formal-language reachability (Definition 3.3) as a graph-theoretic approach to certain database problems. The relevance of CFL-reachability for program analysis was developed in a series of papers by Reps and collaborators [Melski and Reps 2000; Reps 1998; Reps et al. 1995]. It is now applied in many client applications such as shape analysis [Reps 1998], constant propagation [Sagiv et al. 1995], and pointer analysis [Feng et al. 2015].

6.2 InterDyck-Reachability

Dyck-language reachability [Kodumal and Aiken 2004] is a widely used variant of CFL-reachability because Dyck languages can model parenthesis-matching properties such as call/return, lock/unlock, or write-data/read-data. To achieve higher precision, it is natural to use InterDyck-reachability [Li et al. 2020; Reps 2000] for formulating static-analysis problems that model multiple parenthesis-matching properties simultaneously. However, the InterDyck-reachability problem is undecidable in general [Reps 2000]. Many approaches have been proposed to over-approximate InterDyck-reachability, including the three approaches discussed in the paper: CFL-reachability [Reps 1998], SPDS-reachability [Späth et al. 2019], and LCL-reachability [Zhang and Su 2017]. Li et al. [2020] propose a graph-simplification algorithm for InterDyck-reachability. The simplification algorithm can remove edges that do not contribute to InterDyck-paths, thus improving both the performance and the precision of existing InterDyck-reachability algorithms in practice.

Some special cases for InterDyck-reachability are also studied in the literature. Consider the InterDyck-language $L = D_1 \odot D_1$ where D_1 is the Dyck language of one parenthesis type. Its reachability problem on bidirected graphs, where every open-parenthesis edge $u \xrightarrow{(\!|_i} v$ is accompanied by a reverse edge $v \xrightarrow{|\!)_i} u$, has been shown to be in polynomial time [Li et al. 2021]. More recently, the bidirected $D_k \odot D_1$-reachability problem has shown to be decidable, where D_k represents the Dyck language of k different parentheses [Kjelstrøm and Pavlogiannis 2022].

6.3 Parikh Image and Vector Addition Systems with States

Our work formulates the single-source-single-target InterDyck-reachability problem as a parenthesis-counting problem. In formal-language theory, the idea of letter counting can be formulated using the concept of Parikh images [Parikh 1966]. Given a word $w \in \Sigma^*$, where $\Sigma = \{\alpha_1, \cdots, \alpha_k\}$, let $|w|_{\alpha_i}$ denote the number of occurrences of α_i in w. Then the Parikh image of w is a tuple $(|w|_{\alpha_1}, \cdots, |w|_{\alpha_k})$. Parikh images have been extensively studied in the literature.

A seminal result shows that the Parikh images of context-free languages are semilinear sets, and thus they can be encoded using Presburger arithmetic [Parikh 1966]. Later work has focused on providing systematic Presburger-arithmetic encodings of Parikh images [Esparza 1997; Seidl et al. 2004; Verma et al. 2005]. In InterDyck-reachability, the numbers of open parentheses and close parentheses in any valid InterDyck-paths should be the same. Consequently, the Parikh image of the realized string of a valid InterDyck-path should be an all-zeros vector. However, all-zeros Parikh-image reachability does not precisely model the parenthesis-counting formulation of InterDyck-reachability, because InterDyck-reachability requires that there must always be more open parentheses than close parentheses (of a given type) along a valid InterDyck-path. In addition, our algorithm captures the parenthesis ordering by combining ILP solving and the LCL-reachability algorithm. An example illustrating the differences can be found in Example 4.5.

Another related formulation is the reachability problem for vector addition systems with states (VASS) [Hopcroft and Pansiot 1979]. VASS with d counters (d-dimensional VASS) can simulate parenthesis counting for an InterDyck-language consisting of at most d D_1 languages, *i.e.*, reachability for d-dimensional VASS can precisely capture $D_1 \odot D_1 \cdots \odot D_1$-reachability (the number of instances of D_1 is d). VASS-reachability can serve as an over-approximation of a general InterDyck-reachability problem: a language D_k can be approximated as D_1 by treating all open-parenthesis symbols as a single open-parenthesis symbol, and all close-parenthesis symbols as the corresponding single close-parenthesis symbol. By treating each Dyck language this way, a general InterDyck-reachability problem is turned into $D_1 \odot D_1 \cdots \odot D_1$. VASS-reachability has been extensively studied in the literature [Englert et al. 2016; Ganardi et al. 2022; Schmitz 2016]. However, the VASS-reachability problem is well-known to be a hard problem. Even the two-dimensional The VASS-reachability problem is PSPACE-complete [Blondin et al. 2021], and thus a precise algorithm for the VASS-reachability problem is unlikely to be efficient in practice. Note that when discussing the complexity of two-dimensional VASS-reachability, there exist two problem encoding schemas: unary encoding [Englert et al. 2016] and binary encoding [Blondin et al. 2021]. VASS-reachability serves as an over-approximation of InterDyck-reachability in binary encoding.

7 CONCLUSION

This paper has proposed a new single-source-single-target InterDyck-reachability algorithm. Specifically, it formulates InterDyck-reachability as a parenthesis-counting problem over path expressions. We have implemented the algorithm and evaluated it on two client analyses. The experimental results demonstrate that the algorithm is 1.5× faster than the state-of-the-art all-pairs LCL-reachability and can achieve an order-of-magnitude speedup compared to all-pairs SPDS-reachability. Furthermore, it can reduce the number of false-positive node pairs in LCL-reachability and SPDS-reachability algorithms. Finally, it also can achieve better running time compared with two domain-specific analysis tools over the corresponding benchmarks.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback on earlier drafts of this paper. This work was supported, in part, by Amazon under an Amazon Research Award in automated reasoning; by a gift from Rajiv and Ritu Batra; by the Office of Naval Research under grant N00014-17-1-2889; by the US National Science Foundation (NSF) under grants 1917924, 2114627, 2211968, and 2212558; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024; by Facebook Graduate Fellowship for the first author. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsoring entities.

REFERENCES

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

Michael Blondin, Matthias Englert, Alain Finkel, Stefan Göller, Christoph Haase, Ranko Lazic, Pierre McKenzie, and Patrick Totzke. 2021. The Reachability Problem for Two-Dimensional Vector Addition Systems with States. *J. ACM* 68, 5 (2021), 34:1–34:43. https://doi.org/10.1145/3464794

Matthias Englert, Ranko Lazic, and Patrick Totzke. 2016. Reachability in Two-Dimensional Unary Vector Addition Systems with States is NL-Complete. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 477–484. https://doi.org/10.1145/2933575.2933577

Javier Esparza. 1997. Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes. *Fundam. Informaticae* 31, 1 (1997), 13–25. https://doi.org/10.3233/FI-1997-3112

Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9458)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, 465–484. https://doi.org/10.1007/978-3-319-26529-2_25

Moses Ganardi, Rupak Majumdar, Andreas Pavlogiannis, Lia Schütze, and Georg Zetzsche. 2022. Reachability in Bidirected Pushdown VASS. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France (LIPIcs, Vol. 229)*, Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 124:1–124:20. https://doi.org/10.4230/LIPIcs.ICALP.2022.124

E. Goto. 1974. *Monocopy and Associative Algorithms in an Extended LISP.* Technical Report. Information Science Laboratory, Univ. of Tokyo.

Gurobi Optimization, LLC. 2021. Gurobi Optimizer Reference Manual. https://www.gurobi.com

Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. *ACM SIGPLAN Notices* 36, 5 (2001), 24–34.

John E. Hopcroft and Jean-Jacques Pansiot. 1979. On the Reachability Problem for 5-Dimensional Vector Addition Systems. *Theor. Comput. Sci.* 8 (1979), 135–159. https://doi.org/10.1016/0304-3975(79)90041-0

Wei Huang, Yao Dong, Ana L. Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 106–117. https://doi.org/10.1145/2771783.2771803

Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The decidability and complexity of interleaved bidirected Dyck reachability. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–26. https://doi.org/10.1145/3498673

John Kodumal and Alexander Aiken. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William W. Pugh and Craig Chambers (Eds.). ACM, 207–218. https://doi.org/10.1145/996841.996867

Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 780–793. https://doi.org/10.1145/3385412.3386021

Yuanbo Li, Qirun Zhang, and Thomas W. Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434340

Peizun Liu and Thomas Wahl. 2018. CUBA: interprocedural Context-UnBounded Analysis of concurrent programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 105–119. https://doi.org/10.1145/3192366.3192419

David Melski and Thomas W. Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. Comput. Sci.* 248, 1-2 (2000), 29–98. https://doi.org/10.1016/S0304-3975(00)00049-9

Rohit Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (1966), 570–581. https://doi.org/10.1145/321356.321364

T. Reps. 1998. Program Analysis via Graph Reachability. *Inf. and Softw. Tech.* 40, 11–12 (1998), 701–726.

Thomas W. Reps. 2000. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 49–61. https://doi.org/10.1145/199448.199462

Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1995. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 915)*, Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach (Eds.). Springer, 651–665. https://doi.org/10.1007/3-540-59293-8_226

Sylvain Schmitz. 2016. The complexity of reachability in vector addition systems. *ACM SIGLOG News* 3, 1 (2016), 4–21. https://doi.org/10.1145/2893582.2893585

Alexander Schrijver. 1998. *Theory of linear and integer programming.* John Wiley & Sons.

Helmut Seidl, Thomas Schwentick, Anca Muscholl, and Peter Habermehl. 2004. Counting in Trees for Free. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings (Lecture Notes in Computer Science, Vol. 3142)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer, 1136–1149. https://doi.org/10.1007/978-3-540-27836-8_94

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL (2019), 48:1–48:29. https://doi.org/10.1145/3290361

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 59–76. https://doi.org/10.1145/1094811.1094817

Robert Endre Tarjan. 1981. Fast Algorithms for Solving Path Problems. *J. ACM* 28, 3 (1981), 594–614. https://doi.org/10.1145/322261.322273

Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. 2005. On the Complexity of Equational Horn Clauses. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3632)*, Robert Nieuwenhuis (Ed.). Springer, 337–352. https://doi.org/10.1007/11532231_25

Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. 2007. Model checking concurrent Linux device drivers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* 501–504. https://doi.org/10.1145/1321631.1321719

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 155–165. https://doi.org/10.1145/2001420.2001440

Mihalis Yannakakis. 1990. Graph-Theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*, Daniel J. Rosenkrantz and Yehoshua Sagiv (Eds.). ACM Press, 230–242. https://doi.org/10.1145/298514.298576

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 435–446. https://doi.org/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, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 344–358. https://doi.org/10.1145/3009837.3009848

Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 829–845. https://doi.org/10.1145/2660193.2660213

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 197–208. https://doi.org/10.1145/1328438.1328464

Received 2022-07-07; accepted 2022-11-07