Recursive State Machine Guided Graph Folding for Context-Free Language Reachability

YUXIANG LEI, University of New South Wales, Australia
YULEI SUI, University of New South Wales, Australia
SHIN HWEI TAN, Concordia University, Canada
QIRUN ZHANG, Georgia Institute of Technology, USA

Context-free language reachability (CFL-reachability) is a fundamental framework for program analysis. A large variety of static analyses can be formulated as CFL-reachability problems, which determines whether specific source-sink pairs in an edge-labeled graph are connected by a reachable path, i.e., a path whose edge labels form a string accepted by the given CFL. Computing CFL-reachability is expensive. The fastest algorithm exhibits a slightly subcubic time complexity with respect to the input graph size. Improving the scalability of CFL-reachability is of practical interest, but reducing the time complexity is inherently difficult.

In this paper, we focus on improving the scalability of CFL-reachability from a more practical perspective—reducing the input graph size. Our idea arises from the existence of trivial edges, i.e., edges that do not affect any reachable path in CFL-reachability. We observe that two nodes joined by trivial edges can be folded—by merging the two nodes with all the edges joining them removed—without affecting the CFL-reachability result. By studying the characteristic of the recursive state machines (RSMs), an alternative form of CFLs, we propose an approach to identify foldable node pairs without the need to verify the underlying reachable paths (which is equivalent to solving the CFL-reachability problem). In particular, given a CFL-reachability problem instance with an input graph $G$ and an RSM, based on the correspondence between paths in $G$ and state transitions in RSM, we propose a graph folding principle, which can determine whether two adjacent nodes are foldable by examining only their incoming and outgoing edges.

On top of the graph folding principle, we propose an efficient graph folding algorithm $Gf$. The time complexity of $Gf$ is linear with respect to the number of nodes in the input graph. Our evaluations on two clients (alias analysis and value-flow analysis) show that $Gf$ significantly accelerates RSM/CFL-reachability by reducing the input graph size. On average, for value-flow analysis, $Gf$ reduces 60.96% of nodes and 42.67% of edges of the input graphs, obtaining a speedup of $4.65 \times$ and a memory usage reduction of 57.35%. For alias analysis, $Gf$ reduces 38.93% of nodes and 35.61% of edges of the input graphs, obtaining a speedup of $3.21 \times$ and a memory usage reduction of 65.19%.

CCS Concepts: • Theory of computation → Grammars and context-free languages.

Additional Key Words and Phrases: CFL-reachability, recursive state machines, graph simplification

ACM Reference Format:

Authors’ addresses: Yuxiang Lei, yuxiang.lei@unsw.edu.au, University of New South Wales, Australia; Yulei Sui, y.sui@unsw.edu.au, University of New South Wales, Australia; Shin Hwei Tan, shinhwei.tan@concordia.ca, Concordia University, Canada; Qirun Zhang, qrzhang@gatech.edu, Georgia Institute of Technology, USA.
1 INTRODUCTION

Context-free language reachability (CFL-reachability) [Reps 1998], also extensively studied in an alternative form called recursive state machine reachability (RSM-reachability) [Alur et al. 2005a], is a fundamental framework for program analysis. A large variety of static analyses, such as dataflow analysis [Reps et al. 1995], polymorphic flow analysis [Rehof and Fähndrich 2001], typestate analysis [Naeem and Lhoták 2008], code embedding [Sui et al. 2020] and point-to analysis [Zheng and Rugina 2008] can be formulated as CFL-reachability problems, which determine whether specific source-sink pairs in an edge-labeled graph can be connected by a path whose edge labels form a string accepted by the given CFL.

Unfortunately, solving CFL-reachability is quite expensive. The standard algorithm [Melski and Reps 2000; Reps et al. 1995] exhibits a cubic time complexity with respect to the input graph size. Improving the time complexity of CFL-reachability is inherently difficult. Due to Chatterjee et al. [2018], the truly subcubic $O(n^{3-\epsilon})$ bounds of CFL-reachability cannot be obtained without obtaining $O(n^{3-\epsilon})$-time combinatorial algorithms for Boolean Matrix Multiplication [Williams and Williams 2018]. In the literature, asymptotically faster algorithms are known only on restricted context-free languages, i.e., bidirected Dyck-reachability [Chatterjee et al. 2018; Zhang et al. 2013].

Instead of reducing the (slightly) subcubic complexity, we focus on improving the scalability from a more practical perspective—reducing the size of input graphs. In the context of CFL-reachability (and its alternative forms such as set-constraint [Melski and Reps 2000]), several graph simplification techniques have been proposed: (1) The most popular technique is cycle elimination [Fähndrich et al. 1998; Hardekopf and Lin 2007a; Lei and Sui 2019; Pereira and Berlin 2009]. However, there is no general cycle elimination algorithm for CFL-reachability because whether a cycle can be collapsed depends on the context-free grammar (CFG). Moreover, our empirical results (Section 6) show that there are still a large number of collapsible nodes and edges, which do not reside in cycles; (2) Recently, a graph simplification algorithm [Li et al. 2020] was proposed to remove particular “non-Dyck-contributing” parenthesis-labeled edges for problems where the underlying CFL over-approximates the intersection of multiple Dyck languages (i.e., the INTERDYCK language). Nevertheless, there are a variety of problems where the parenthesis-labeled edges do not dominate [Naeem and Lhoták 2008; Sui and Xue 2016a; Zheng and Rugina 2008], whereby the effectiveness of the INTERDYCK graph simplification algorithm compromises; and (3) Edge contraction for particular clients has also been studied. The best-known method is the offline variable substitution for pointer analysis [Rountev and Chandra 2000], a special case of the more general graph folding problem.

In this paper, we introduce a novel graph folding technique for CFL-reachability. The graph folding technique extends the applicability of edge contraction from the special case [Rountev and Chandra 2000] to general CFL-reachability. The foldability of CFL-reachability originates from the existence of trivial edges. If two nodes are joined by trivial edges, they can be folded—by merging the two nodes and removing all the edges joining them—without affecting the CFL-reachability result.

The challenge lies in identifying foldable node pairs. Intuitively, we can identify the foldable node pairs by determining all reachable paths. However, this is equivalent to solving CFL-reachability and obviously defeats the purpose of improving scalability. In this paper, we utilize an equivalent form of CFLs called recursive state machines (RSMs) and address the challenge for problems where the CFLs can be expressed as deterministic RSMs. In deterministic RSMs, each target state can be uniquely determined by the source state and the transition label. Therefore, our key technical insight is to study the correspondence between paths in graphs and state transition chains in deterministic RSMs, and refine the criteria for foldable node pairs. In particular, we consider a node pair $(x, y)$ as foldable if the corresponding RSM transitions of each path in the graph are consistent before and after folding $(x, y)$. By further exploiting the dependency of RSM state transitions, we
introduce a graph folding principle for deterministic RSMs, which can identify foldable node pairs by examining only their incoming and outgoing edges in the input graph.

On top of the graph folding principle, we propose a graph-folding algorithm $G_f$, which has a linear time complexity with respect to the number of nodes in the input graph. $G_f$ enables a general preprocessing step for CFL/RSM-reachability where the RSM is deterministic. Deterministic RSMs cover a large variety of languages extensively used in program analysis, including visibly pushdown languages [Alur and Madhusudan 2004], Dyck languages [Kodumal and Aiken 2004], and all regular languages. By reducing the input graph size, graph folding significantly improves the scalability of CFL-reachability in terms of both time and space.

We have implemented $G_f$ and applied it to a context-sensitive value-flow analysis [Sui and Xue 2018; Sui et al. 2014], and a field-sensitive alias analysis [Zheng and Rugina 2008]. We have extensively evaluated it using benchmarks of ten popular GitHub projects in C/C++. Our experimental results show that $G_f$ can significantly reduce the graph sizes for the two client analyses. In general, by reducing 60.96% nodes and 42.67% edges of the input graphs, $G_f$ accelerates context-sensitive value-flow analysis by 4.65× with a memory reduction rate of 57.35%; by reducing 38.93% nodes and 35.61% edges, $G_f$ accelerates field-sensitive alias analysis by 3.21× with a memory reduction rate of 65.19%. The performance of $G_f$ in both clients surpasses existing graph simplification techniques [Li et al. 2020; Nuutila and Soisalon-Soininen 1994].

This paper makes the following contributions:

1. We introduce a new perspective that utilizes deterministic RSMs to simplify the input graphs of CFL-reachability.
2. For deterministic CFL- and RSM-reachability, we propose a graph folding principle that identifies foldable node pairs by examining only their incoming and outgoing edges in the input graph.
3. We propose an efficient graph-folding algorithm $G_f$, whose time complexity is linear with respect to the number of nodes of the input graph.
4. We apply $G_f$ to a context-sensitive value-flow analysis and a field-sensitive alias analysis on ten open-source C/C++ programs. The results demonstrate that graph folding significantly reduces running time and memory overhead by reducing the input graph size.

The remainder of this paper is organized as follows: Section 2 presents a motivating example. Section 3 introduces the background and formulates graph folding. Section 4 studies the criteria for foldable node pairs. Section 5 proposes our graph-folding algorithm $G_f$, followed by experiments in Section 6, and related work and conclusion in Sections 7 and 8, respectively.

2 MOTIVATING EXAMPLE

This section motivates graph folding for CFL-reachability, describing the benefits and challenges. Figure 1(a) shows a CFG of a Dyck language variant and its corresponding RSM. Let $S$ be the start symbol of the CFG in Figure 1(a). This grammar generates a Dyck language of matched parentheses “(” and “)” with an arbitrary number of “a”s. Dyck languages are widely used in context-sensitive program analyses [Hao et al. 2015; Heine and Lam 2003; Reps et al. 1995; Zhang and Su 2017]. In this motivating example, we focus on the reachability problem that identifies $S$-reachable node pairs starting with $v_0$ and $v_1$. Figure 1(b) gives the input graph $G$. In $G$, there are four $S$-reachable node pairs $(v_0, v_7), (v_1, v_5), (v_1, v_6)$ and $(v_1, v_7)$. Figure 1(c) gives the details of the $S$-reachable node pairs and the corresponding $S$-paths, where the column “String” contains the corresponding path strings. Note that there is no cycle in $G$, and removing any parenthesis-labeled edge changes the CFL-reachability solution. Hence, neither cycle elimination [Nuutila and Soisalon-Soininen 1994] nor InterDyck graph simplification [Li et al. 2020] can simply $G$ in this example.
Graph Folding. Graph folding reduces the graph size by merging adjacent node pairs and removing the corresponding edges. Most importantly, it does not affect the original reachability result. Figure 1(b) gives the folded graph $G'$ from $G$. Specifically, the node pair $(v_2, v_3)$ is folded into a representative node $v'_2$ with $v_2 \stackrel{a}{\rightarrow} v_3$ removed. Likewise, $(v_2, v_4)$ and $(v_5, v_6)$ are folded into $v'_2$ and $v'_3$, respectively. Figure 1(d) gives the reachability result on $G'$. Notably, $v'_2$ in $G'$ represents nodes $v_5$ and $v_6$ in $G$. By expanding $v'_2$ in $(v_1, v'_2)$ back into $v_5$ and $v_6$, we get two $S$-reachable pairs $(v_1, v_5)$ and $(v_1, v_6)$. As a result, the reachability results from $G'$ and $G$ are the same. Hence, graph folding can reduce the size of $G$ while maintaining the correctness of the solution.

Benefits. Proper preprocessing that shrinks the input graph can significantly reduce the computation overhead for CFL-reachability. Given a graph $G$, standard CFL-reachability algorithm [Melski and Reps 2000] takes cubic time with respect to the number of nodes in $G$. Specifically, in Figure 1(b), there are eight nodes and eight edges in the original graph $G$. Assume that each iteration inserts a summary edge into the graph. Then, performing the standard CFL-reachability algorithm upon $G$ costs 23 iterations. In contrast, there are only five nodes and five edges in the folded graph $G'$, which only costs 13 iterations to obtain the result. By reducing the graph size, graph folding can reduce 43.48% of iterations. Moreover, our graph-folding algorithm Gr has a linear time complexity with respect to the number of nodes in the graph, which is asymptotically faster than any CFL-reachability algorithm. Specifically, Gr folds the graph $G$ in the motivating example in eight iterations. Even if we take Gr running time into account, the overall computation overhead is smaller than that without any preprocessing.

Challenge. Identifying foldable node pairs is the main technical challenge of graph folding. From the motivating example in Figure 1, we can see that we cannot fold arbitrary node pairs in $G$. Consider the node pair $(v_3, v_5)$. If we fold the pair into a representative node $v'_3$, the path

---

**Fig. 1. Motivating example for graph folding.**

<table>
<thead>
<tr>
<th>Solution</th>
<th>Reachable path</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v_0, v_7)$</td>
<td>$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow b \rightarrow v_5 \rightarrow a \rightarrow v_7$</td>
<td>$(a)$</td>
</tr>
<tr>
<td>$(v_1, v_5)$</td>
<td>$v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow v_5$</td>
<td>$(a)$</td>
</tr>
<tr>
<td>$(v_1, v_6)$</td>
<td>$v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow v_5 \rightarrow a \rightarrow v_6$</td>
<td>$(a)$</td>
</tr>
<tr>
<td>$(v_1, v_7)$</td>
<td>$v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow v_5 \rightarrow b \rightarrow v_7$</td>
<td>$(a)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
<th>Reachable path</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v_0, v_7')$</td>
<td>$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow b \rightarrow v_5 \rightarrow a \rightarrow v_7$</td>
<td>$(a)$</td>
</tr>
<tr>
<td>$(v_1, v_5')$</td>
<td>$v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow v_5$</td>
<td>$(a)$</td>
</tr>
<tr>
<td>$(v_1, v_6')$</td>
<td>$v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow v_5 \rightarrow a \rightarrow v_6$</td>
<td>$(a)$</td>
</tr>
<tr>
<td>$(v_1, v_7')$</td>
<td>$v_1 \rightarrow v_2 \rightarrow a \rightarrow v_3 \rightarrow v_5 \rightarrow b \rightarrow v_7$</td>
<td>$(a)$</td>
</tr>
</tbody>
</table>
When discussing graph reachability, this makes the correspondence between paths in the graph and transition chains in RSMs more intuitive. For instance, a labeled edge \( v_1 \xrightarrow{t} v_2 \) in the input graph can be directly mapped to a state transition \( s_1 \xrightarrow{t} s_2 \) in the RSM, where \( s_1 \) and \( s_2 \) are two RSM states. This section introduces the preliminaries and formulates the problem.

### 3.1 Recursive State Machines

A recursive state machine (RSM) \cite{Alur2005a} over a finite alphabet \( \Sigma \) is defined as a tuple \( R = \langle M_1, \ldots, M_t \rangle \) comprised of \( t \) component finite state machines, where each component \( M_i = \langle N_i, B_i, Y_i, \text{En}_i, \text{Ex}_i, \delta_i \rangle \) is a finite state machine consisting of the following:

- \( N_i \): a finite set of local states.
- \( B_i \): a finite set of boxes in \( M_i \), with each of which mapped to a component state machine.
- \( Y_i : B_i \mapsto \{1, \ldots, t\} \): a mapping function assigning each box of \( M_i \) an index of one of the components \( M_1, \ldots, M_t \).
- \( \text{En}_i \subseteq N_i \): a set holding the entries of \( M_i \).
- \( \text{Ex}_i \subseteq N_i \): a set holding the exits of \( M_i \).
- \( \delta_i : (N_i \cup \bigcup_{b \in B_i} \text{Ex}_{Y_i(b)}) \times \Sigma \rightarrow N_i \cup \bigcup_{b \in B_i} \text{En}_{Y_i(b)} \): a local transition function that maps specific states and labels to specific target states.

Specifically, for a local transition in \( \delta_i \), denoted by \( n_i \xrightarrow{t} n_{i_2} \), (1) the source \( n_i \) must be either a local state or an exit of a box belonging to \( M_i \); (2) the label \( t \) is an element of \( \Sigma \), and (3) the target \( n_{i_2} \) must be either a local state or an entry of a box in \( M_i \).

**Example 3.1.** Figure 1(a) gives an example RSM over an alphabet \( \Sigma = \{\langle, a, \rangle\} \). There is only one component \( M_1 \) calling itself recursively. \( M_1 \) is comprised of: (1) a local state \( n_1 \), which is also the entry and the exit of \( M_1 \); (2) a box \( b_1 \) which is mapped to \( M_1 \); and (3) three transition rules \( n_1 \xrightarrow{a} \langle b_1, n_1 \rangle, n_1 \xrightarrow{a} n_1 \) and \( \langle b_1, n_1 \rangle \xrightarrow{a} n_1 \) where \( \langle b_1, n_1 \rangle \) denotes the entry (also the exit) of the box \( b_1 \), as the labeled edges in Figure 1(a).

The semantics of RSM is given by global states and transitions:

**Global states.** A global state \( s \in S \), where \( S = B^* \times N, B = \bigcup_i B_i, N = \bigcup_i N_i \), can be viewed as a local state nested in layers of boxes. Consider a sequence of boxes \( b_1, \ldots, b_k \) such that \( b_i \in B_{i-1} \) and \( B_{j_i}, B_{j_{i+1}} \in B \) for all \( i \in \{1, \ldots, k\} \). A global state \( s = \langle b_1, \ldots, b_k, n \rangle \) follows (1) \( Y_{j_i}(b_i) = j_{i+1} \) for all \( i \in \{1, \ldots, k\} \), i.e., \( b_{j_k} \) is nested in \( b_k \) and (2) \( n \notin N_{j_k} \), i.e., \( n \) is nested in \( b_k \). Intuitively, a global state denotes the current position of the initial state (usually not nested in any box) after a series of transitions, including entering/exiting boxes.
Global transitions $\Delta$. The global transition function $\Delta : S \times \Sigma \rightarrow S$ maps specific global states and labels to specific target global states. A global transition rule is denoted by $s_1 \xrightarrow{\ell} s_2 \in \Delta$, where $s_1, s_2 \in S$ and $\ell \in \Sigma$. Global transitions are restricted by the local transition rules. Given two global states $s_1, s_2 \in S$ and $s_1 = (b_1, \cdots, b_{k-1}, b_k, n_1)$ where $b_k \in B_{j_k}$, $Y_{j_k}(b_k) = j_{k+1}$ and $n_1 \in N_{j_{k+1}}$, $s_1 \xrightarrow{\ell} s_2 \in R$ iff one of the following four situations holds:

1. Location transition: $s_2 = (b_1, \cdots, b_{k-1}, b_k, n_2)$ and $n_1 \xrightarrow{\ell} n_2 \in \delta_{j_{k+1}}$;
2. Entering a box: $s_2 = (b_1, \cdots, b_{k-1}, b_k, n_2)$, $b_k \in B_{j_{k+1}}$, $n_2 \in En_{j_{k+1}}(b_{k+1})$ and $n_1 \xrightarrow{\ell} (b_{k+1}, n_2) \in \delta_{j_{k+1}}$;
3. Exiting from a box: $s_2 = (b_1, \cdots, b_{k-1}, n_2)$, $n_1 \in Ex_{j_{k+1}}$ and $(b_k, n_1) \xrightarrow{\ell} n_2 \in \delta_{j_k}$;
4. Transiting across two boxes: $s_2 = (b_1, \cdots, b_{k-1}, b'_k, n_2)$, $n_1 \in Ex_{j_{k+1}}$, $b'_k \in B_{j_{k}}$, $n_2 \in En_{j_{k}}(b'_k)$ and $(b_k, n_1) \xrightarrow{\ell} (b'_k, n_2) \in \delta_{j_k}$.

Property 3.1 (Dependency of Global Transitions on Local Transitions). A global transition can only change the innermost box (by entering/exiting) and the innermost local state (by the local transition function of the component state machine, to which the innermost box maps) of a global state.

For brevity, we refer to all the states and transitions as the global ones unless otherwise specified.

Deterministic RSMs. In a deterministic RSM $R$, given a state $s_I$ and a label $\ell$, if $\exists s_J \xrightarrow{\ell} s_J \in R$, $s_J$ is unique. In other words, for the transition function of a deterministic RSM, when the input state and label are specified, we can always determine the output state. In the remaining sections, all demonstrations and conclusions are based on deterministic RSMs.

Transition Chains. In an RSM $R$, a transition chain $p_R \in R$ from a source state $s_0$ to a target state $s_k$ is a sequence of global transitions $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_k} s_k$ such that $s_{i-1} \xrightarrow{\ell_i} s_i \in \Delta$ for all $i \in \{1, \cdots, k\}$. In a deterministic RSM, given a source state $s_0$ and a sequence of label $\ell_1, \cdots, \ell_k$, the transition chain $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_k} s_k \in R$ is unique if it exists.

Strings Accepted by an RSM. Given an RSM $R$ with a specified initial state $s_{init} \in S$ and a set of accepting states $F \subseteq S$, a string $w \in \Sigma^*$ is accepted by $R$ iff it takes a transition chain from the initial state to one of the accepting states, i.e., $\exists s_{init} \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} s_k \in R$ such that $s_k \in F$ and $\ell_1 \cdots \ell_k = w$.

3.2 RSM-Reachability

Given a source-sink node pair, RSM-reachability is to check whether the sink is reachable from the source by a path whose edge labels form, in sequence, a string accepted by the RSM.

Reachable Paths and Pairs. Given an RSM-reachability instance $Reach(R, G)$ where $R$ is an RSM and $G = (V, E)$ is an edge-labeled directed graph, a path $p_G = v_i \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} v_j \in G$ is a reachable path iff the sequence of the edge labels $\ell_1 \cdots \ell_k$ forms a string accepted by $R$. In an RSM-reachability instance, a reachable pair $(v_i, v_j) \in V \times V$ is a node pair in $G$ such that there exists at least one reachable path from $v_i$ to $v_j$.

RSM-reachability. Formally, given an RSM $R$ with a specified initial state and accepting states and a graph $G$ with specified sources $V_{src} \subseteq V$ and sinks $V_{sink} \subseteq V$, an RSM-reachability problem aims to determine for each source-sink pair $(v_i, v_j) \in V_{src} \times V_{sink}$ whether it is a reachable pair.

Example 3.2. We formulate the motivating example (Figure 1) as an RSM-reachability problem. In this instance, $R = (M_1), s_{init} = (n_1), F = \{\langle n_1 \rangle\}, V_{src} = \{v_0, v_1\}, V_{sink} = \{v_0, \cdots, v_7\}$. For the path...
\[ v_1 \xrightarrow{a} v_2 \xrightarrow{a} v_3 \xrightarrow{a} v_5 \in G, \text{ the sequence of edge labels forms a string } \langle a \rangle \text{ which is accepted by } R \]

because there is a transition chain \( \langle n_1 \rangle \xrightarrow{a} \langle b_1, n_1 \rangle \xrightarrow{a} \langle b_1, n_1 \rangle \xrightarrow{1} \langle n_1 \rangle \in R \). Therefore, \((v_1, v_5)\) is a reachable pair. Similarly, \((v_0, v_7), (v_1, v_6)\) and \((v_1, v_7)\) are also reachable pairs.

### 3.3 Graph Folding

Given a directed graph \( G \) and two adjacent nodes \((x, y) \in V \times V\), an \( xy \)-folded graph \( G' \) is constructed by removing all the edges joining \( x \) and \( y \) and collapsing \((x, y)\) into a representative node \( z \) such that \( \text{Rep}(x) = \text{Rep}(y) = z \). In particular, for a node \( v_i \in V \), if \( v_i \) is merged into another node \( z \), \( \text{Rep}(v_i) = z \), otherwise, \( \text{Rep}(v_i) = v_i \).

Intuitively, an ideal graph folding approach should fold as many node pairs as possible. However, arbitrarily folding node pairs can change reachable pairs due to removals or additions of graph edges, resulting in an incorrect RSM-reachability solution. A correct graph folding approach must guarantee the equivalence of reachable pairs in the original graph and the folded graph (Definition 3.1).

**Definition 3.1 (Reachability Equivalence).** Let \( G = \langle V, E \rangle \) be the original graph with \( V_{src} \) and \( V_{snk} \) specified, and let \( G' = \langle V', E' \rangle \) be the folded graph, \( G \) and \( G' \) are reachability equivalent iff \( (v_i, v_j) \in V_{src} \times V_{snk}, v_j \) is reachable from \( v_i \) in \( G \) iff \( \text{Rep}(v_j) \) is reachable from \( \text{Rep}(v_i) \) in \( G' \).

Definition 3.1 implies that a correct graph folding preserves the information of all the original reachable pairs. In other words, two graphs being reachability equivalent yields equivalent RSM-reachability solutions. If folding a node pair \((x, y)\) preserves reachability equivalence, we say that \((x, y)\) is foldable.

We formulate our graph folding problem as follows.

Given an RSM-reachability instance \( \text{Reach}(R, G) \), generate a smaller graph \( G' \) by folding node pairs in \( G \) and guarantee that \( G \) and \( G' \) are reachability equivalent.

### 4 PRINCIPLE FOR GRAPH FOLDING

Identifying foldable node pairs based on Definition 3.1 requires computing all reachable paths in the original graph \( G \) and the folded graph \( G' \), which contradicts the goal of improving scalability for RSM-reachability. In this section, we study the correspondence between nodes in the graph and states in the RSM. Specifically, we show that when folding a node pair \((x, y)\), whether reachability equivalence is preserved can be determined by at most two transitions, starting with the corresponding states of \( x \) and \( y \) and going through the edge labels involving \( x \) and \( y \). We formulate our folding principle by overapproximating the corresponding states of \( x \) and \( y \) and exploiting RSM properties, i.e., the subsumption and equivalence relations of states. Thus, we can efficiently identify whether a node pair is foldable by examining only its incoming and outgoing edges.

#### 4.1 Correspondence between Graph Folding and RSM-Reachability

**Folding-Equivalent Class and Criteria for Reachability Equivalence.** Consider Figure 2(a). The path \( p_G \) is obtained from \( p_G' \) by folding \((x, y)\), i.e., contracting the edge \( x \xrightarrow{\epsilon} y \). In fact, the input graph in RSM-reachability can be a multigraph, meaning that there can be multiple edges between \( x \) and \( y \), as shown in Figure 2(b). As folding \((x, y)\) removes all the edges joining \( x \) and \( y \), there can be multiple original paths folded into the same \( xy \)-folded path. Such paths in the original graph \( G \) constitute folding-equivalent classes based on their endpoints.
where we will provide an approach to identify foldable pairs that meet these conditions.

As depicted in Figure 119:8 Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang

Definition 4.1 (xy-Folding-Equivalent Classes). Two paths \( p_1, p_2 \in G \) are in the xy-folding-equivalent (xy-FEQ) class \( P_{xy} \) iff they have identical endpoints and are folded into the same path \( p' \) in the xy-folded graph \( G' \). A path is xy-FEQ to itself.

In Figure 2(b), all the paths starting with \( v_1 \) and ending with \( v_2 \) are in the xy-FEQ class. Based on xy-folded paths and their corresponding xy-FEQ classes, we state sufficient conditions for graph folding that can guarantee reachability equivalence in Definition 4.2. Note that our sufficient conditions do not rely on obtaining all reachable node pairs.

Definition 4.2 (Sufficient Conditions for Reachability Equivalence). An xy-folded graph \( G' \) is reachability equivalent to its original graph \( G \) if the following conditions are satisfied:

Cond. 1. Exclusiveness: Each source-sink path in \( G' \) has a corresponding \( xy \)-FEQ class \( P_{xy} \) in \( G \).

Cond. 2. Consistency: For each \( xy \)-FEQ class \( P_{xy} \) in \( G \), its corresponding \( xy \)-folded path in \( G' \) is a reachable path iff \( P_{xy} \) contains a reachable path in \( G \).

Satisfying the exclusiveness condition (Cond. 1) is straightforward. For example, we can avoid introducing extra source-sink paths to \( G' \) by not merging \( (x, y) \) when (1) there is no edge from \( y \) to \( x \) and (2) \( y \) is a source or \( y \) has incoming edges not from \( x \). To satisfy the consistency condition (Cond. 2), we need to check the corresponding transition chains of reachable paths. In Section 4.2, we will provide an approach to identify foldable pairs that meet these conditions.

Corresponding Transition Chains and Corresponding States. As discussed in Section 3.1, given a specific initial state \( s_{init} \) and a string of labels \( \ell_1 \cdots \ell_k \), a deterministic RSM has at most one transition chain from \( s_{init} \), through \( \ell_1 \cdots \ell_k \), to a deterministic target state \( s_k \). Notably, the corresponding transition chain is defined below:

Definition 4.3 (Corresponding Transition Chain). Given an RSM-reachability instance \( \text{Reach}(R, G) \) where \( R \) is deterministic, consider a path \( p_G = v_0 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} v_k \in G \) where \( \ell_1 \cdots \ell_k \) forms a string \( w \in \Sigma^k \). If there exists \( p_R = s_{init} \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} s_k \in R \) such that \( \ell_1 \cdots \ell_k = w \), we say that \( p_R \) is the corresponding transition chain of \( p_G \). As \( p_R \), if it exists, is unique in the deterministic RSM \( R \), a path \( p_G \in G \) has zero or one corresponding transition chain.

Figure 3(a) shows a path \( p_G \) and its corresponding transition chain \( p_R \). Similarly, Figure 3(b) depicts an \( xy \)-folded path \( p_G' \in G' \) which has zero or one corresponding transition chain \( p_R' \in R \). As depicted in Figure 3(a), with respect to the corresponding transition chain, each node \( v_i \) of a path \( p_G \in G \) is mapped to exactly one state \( s_i \) in the RSM. We call \( s_i \) the corresponding state of \( v_i \) in \( G \). In a graph, a node \( v_i \) may belong to multiple paths corresponding to different transition chains, leading to the notion of corresponding states:
Before folding \((x, y)\):

\[
p_G \in G : v_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{k-1}} x \xrightarrow{t_k} y \xrightarrow{t_{k+1}} v_2 \xrightarrow{t_2} \cdots \xrightarrow{t_k} v_k
\]

\[
p_R \in R : s_{init} \xrightarrow{t_0} \cdots \xrightarrow{t_{k-1}} s_x \xrightarrow{t_k} s_y \xrightarrow{t_{k+1}} s_2 \xrightarrow{t_2} \cdots \xrightarrow{t_k} s_k
\]

(a) A path \(p_G\) in \(G\) and its corresponding transition chain \(p_R\) in the RSM \(R\).

After folding \((x, y)\):

\[
p'_G \in G' : v_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{k-1}} x \xrightarrow{t_k} v_2 \xrightarrow{t_2} \cdots \xrightarrow{t_k} v_k
\]

\[
p'_R \in R : s_{init} \xrightarrow{t_0} \cdots \xrightarrow{t_{k-1}} s_x \xrightarrow{t_k} s_y' \xrightarrow{t_{k+1}} s_2 \xrightarrow{t_2} \cdots \xrightarrow{t_k} s_k'
\]

(b) The \(xy\)-folded path \(p'_G\) of \(p_G\) and its corresponding transition chain \(p'_R\).

Fig. 3. A path \(p_G\) and its corresponding transition chain \(p_R\) before and after folding \((x, y)\). Without loss of generality, we assume that \(y\) is merged into \(x\) in the \(xy\)-folded graph.

**Table 1. Edge label notations for discussing RSM-reachability.**

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_{xy}) = ({t \mid x \xrightarrow{t} y \in E})</td>
<td>the set of labels of the edges from (x) to (y).</td>
</tr>
<tr>
<td>(L_{x-}) = ({t \mid x \xleftarrow{t} v_i \in E, v_i \in V})</td>
<td>the set of labels of the outgoing edges of (x).</td>
</tr>
<tr>
<td>(L_{-x}) = ({t \mid v_i \xrightarrow{t} x \in E, v_i \in V})</td>
<td>the set of labels of the incoming edges of (x).</td>
</tr>
<tr>
<td>(L_{fx}) = ({t \mid v_i \xrightarrow{t} x \in E, v_i \in V, v_i \neq y})</td>
<td>the set of labels of the edges ending with (x) and not starting with (y).</td>
</tr>
<tr>
<td>(L_{x-}) = ({t \mid x \xleftarrow{t} v_i \in E, v_i \in V, v_i \neq y})</td>
<td>the set of labels of the edges starting with (x) and not ending with (y).</td>
</tr>
</tbody>
</table>

**Definition 4.4 (Corresponding States).** For a node \(v_i \in V\), its corresponding state \(Q_{v_i} \subseteq S\) is a set that holds the target states of all the corresponding transition chains of paths ending at \(v_i\). Taking empty paths into consideration, we have \(s_{init} \in Q_{v_i}\).

Note that a path \(p_G \in G\) can be a subpath of other paths, which means that \(p_G\) can also correspond to one or more sub-transition chains that do not start with \(s_{init}\). Nevertheless, the starting states of the sub-transition chains must belong to \(Q_{v_i}\).

**Remark (Sub-Transition Chain).** For a path \(p_G\) starting with a node \(v_i\) and being a subpath of other paths in \(G\), each corresponding sub-transition chain of \(p_G\) must start with a state belonging to \(Q_{v_i}\).

**Example 4.1 (Corresponding States and Reachability Equivalence).** Consider an instance in Figure 3 where \(v_2, \ldots, v_k\) do not contain any of \(x\) or \(y\). According to Definition 4.4, \(s_x \in Q_x\). By comparing Figures 3(a) and 3(b), we can see that \(s_2\) is obtained from \(s_x\) by two transitions \(s_x \xrightarrow{\ell_x y} s_y \xrightarrow{\ell_y} s_2\), while \(s'_2\) is obtained from \(s_x\) by one transition \(s_x \xrightarrow{\ell_x} s'_2\). If \(s_2 = s'_2\), then \(s_k = s'_k\) because the sequence of edge labels \(\ell_2, \ell_3, \ldots, \ell_k\) is not changed after folding \((x, y)\). Moreover, if for any \(s_x \in Q_x, s_2 = s'_2\), then for any \(p_G\) (Figure 3(a)) and its \(xy\)-folded \(p'_G\) (Figure 3(b)), their corresponding transition chains end at the same state. This means that folding \((x, y)\) keeps the paths passing through \(x \xrightarrow{\ell_x y} y\) and their \(xy\)-folded paths consistent with respect to whether they are reachable paths.

The above example shows that, with determined \(Q_x\) and \(Q_y\), we are able to identify whether folding \((x, y)\) preserves reachability equivalence by computing at most two transitions for each state of \(Q_x\) and \(Q_y\).
4.2 Folding Principle

Precisely computing $Q_x$ and $Q_y$ is equivalent to solving RSM-reachability, which is too expensive for graph folding. This section formulates our graph folding principle overapproximates $Q_x$, utilizing RSM properties, i.e., subsumption and equivalence relations of states.

Overapproximating $Q_x$. With determined labels of incoming edges of $x$, the local states of the states in $Q_x$ are determined. To facilitate the discussion, Table 1 lists the notations about the edge labels involving $x$ and $y$. With all incoming edge labels $L_x$ of a node $x$ in $G$, we collect a set of RSM local states $n'$ with transitions $n \xrightarrow{\ell} n'$ and $\ell \in L_x$. We define the set $\text{Nr}(L_x)$ as

$$\text{Nr}(L_x) = \{n' \mid n \xrightarrow{\ell} n' \in \bigcup_i \delta_i, \ell \in L_x\}. \quad (1)$$

Intuitively, $\text{Nr}(L_x)$ holds the target RSM states whose transition labels belong to the incoming edge label set $L_x$ of $x$. In particular, if $x \in \text{Vsr}$, we let $s_{\text{init}} \in \text{Nr}(L_x)$. Obviously, the local states of the states in $Q_x$ belong to $\text{Nr}(L_x)$. Computing $\text{Nr}(L_x)$ is very inexpensive because given $L_x$, $\text{Nr}(L_x)$ can be directly determined by checking the local transition functions of the RSM. As global states are local states wrapped by layers of boxes, we have:

$$Q_x \subseteq B^* \times \text{Nr}(L_x). \quad (2)$$

In Section 4.1, we use Example 4.1 to show that we can determine whether folding $(x, y)$ preserves reachability equivalence by computing at most two transitions for each state of $Q_x$ and $Q_y$. According to Property 3.1, each transition can exit at most one box. Namely, for $Q_x$ and $Q_y$, examining the states with two layers of boxes is sufficient to cover all states, as outer boxes are never affected by the two transitions. Correspondingly, while replacing $Q_x$ by $B^* \times \text{Nr}(L_x)$ and $B^* \times \text{Nr}(L_y)$, examining the states of $B^\alpha \times \text{Nr}(L_x)$ and $B^\alpha \times \text{Nr}(L_y)$ where $\alpha \leq 2$ is enough.

Rules for Consistency. In Section 4.1, we show that ensuring reachability equivalence is to satisfy the two conditions of Definition 4.2, among which satisfying the exclusiveness condition (Cond. 1) is simple and addressed immediately after Definition 4.2. Here, we provide four rules in Figure 5 for satisfying the consistency condition (Cond. 2). The four rules exploit subsumption and equivalence relations of states (Definition 4.5). Basically, the two relations are used to measure and compare the capabilities of states being transited by edge labels involving $x$ and $y$.

Definition 4.5 (Subsumption $\leq$ and Equivalence $\simeq$ Relations of States). Given a set of labels $L \subseteq \Sigma$ and two states $s_i, s_j \in S$, $s_i$ is subsumed by $s_j$ with respect to $L$, denoted by $s_i \leq_s s_j$, iff

1. $\forall \ell \in L, \forall s_k \in S, s_i \xrightarrow{\ell} s_k \in \Delta \Rightarrow s_j \xrightarrow{\ell} s_k \in \Delta$, and
2. $s_i \in F \Rightarrow s_j \in F$.

Specifically, $s_i$ is equivalent to $s_j$ with respect to $L$, denoted by $s_i \simeq_s s_j$, iff $s_i \leq_s s_j \land s_j \leq_s s_i$. 

Rule [x-x]: \( \forall s_x \xrightarrow{\ell_{xy}} s_y \xrightarrow{\ell_{yx}} s'_x \in R \text{ s.t. } s_x \in B^a \times Nr(L_x) \land \ell_{xy} \in L_{xy} \land \ell_{yx} \in L_{yx}, \ s'_x \leq s_x. \)

Rule [y-y]: \( \forall s_y \xrightarrow{\ell_{xy}} s_x \xrightarrow{\ell_{yx}} s'_y \in R \text{ s.t. } s_y \in B^a \times Nr(L_y) \land \ell_{xy} \in L_{xy} \land \ell_{yx} \in L_{yx}, \ s'_y \leq s_y. \)

Rule [x-y]: \( \forall s_x \in B^a \times Nr(L_x) \land \forall \ell_{xy} \in L_{xy}, \ \exists s_y \xrightarrow{\ell_{xy}} s_y \in R \text{ s.t. } s_x \xrightarrow{\ell_{yx}} s'_x \in Nr(L_y), \ s'_y \leq s_y. \)

Rule [y-x]: \( \forall s_y \in B^a \times Nr(L_y) \land \forall \ell_{yx} \in L_{yx}, \ \exists s_x \xrightarrow{\ell_{yx}} s_x \in R \text{ s.t. } s_y \xrightarrow{\ell_{xy}} s'_y \in Nr(L_x), \ s'_x \leq s_x. \)

Fig. 5. Rules for Cond. 2, where \( a \in \{0, 1, 2\} \), \( L_{xy}, L_{x-}, L_{y+} \), and \( Nr(L_x) \) are defined in Table 1 and Eq. 1.

Example 4.2. Figure 4(b) gives an example of subsumption and equivalence relations of states in the RSM of Figure 4(a). \( \langle n_0 \rangle \stackrel{L_1}{\not\simeq} \langle n_1 \rangle \) because both \( \langle n_0 \rangle \) and \( \langle n_1 \rangle \) can transit to \( \langle n_2 \rangle \) via \( \ell_2 \). \( \langle n_1 \rangle \stackrel{L_2}{\leq} \langle n_0 \rangle \) because \( \langle n_1 \rangle \) can transit to \( \langle n_2 \rangle \) via \( \ell_2 \) and \( \langle n_0 \rangle \) can not only transit to \( \langle n_2 \rangle \) via \( \ell_2 \) but also transit to \( \langle n_1 \rangle \) via \( \ell_1 \). In contrast, \( \langle n_0 \rangle \not\stackrel{L_2}{=} \langle n_1 \rangle \) because \( \langle n_0 \rangle \) can transit to \( \langle n_1 \rangle \) via \( \ell_1 \) whereas \( \langle n_1 \rangle \) cannot.

Based on the two relations in Definition 4.5, we briefly discuss the rules in Figure 5. By examining the states of \( B^a \times Nr(L_x) \) and the labels of incoming and outgoing edges of \( x \) and \( y \), Rule [x-x] ensures the consistency of the corresponding transition chains of paths passing through \( x \) and \( y \) by starting and ending both with \( x \). Rule [x-y] ensures the consistency of the corresponding transition chains of paths passing through an edge from \( x \) to \( y \). Rules [y-y] and [y-x] are symmetric to Rules [x-x] and [x-y] with respect to \( x \) and \( y \).

Graph Folding Principle. We provide our principle for identifying foldable \((x, y)\) in Theorem 4.1, where Principles 1 and 2 are used to satisfy Cond. 1 and Cond. 2 of Definition 4.2, respectively. Notably, in our folding principle, each rule in Figure 5 contains no more than two transitions, and the value of \( a \), i.e., the layers of boxes, is no more than two.

**Theorem 4.1 (Graph Folding Principle).** Consider an RSM-reachability instance \( \text{Reach}(R, G) \). Without loss of generality, assume that there is always at least one edge from node \( x \) to \( y \) in \( G \). The node pair \((x, y)\) is foldable if both 1 and 2 hold:

1. When there is no edge from \( y \) to \( x \), \( y \notin V_{src} \) and all the incoming edges of \( y \) starts at \( x \);
2. The four rules in Figure 5 hold for all \( a \in \{0, 1, 2\} \).

Example 4.3 (Identifying Foldable Node Pairs). Figure 4(a) is an RSM, and Figures 4(c) and (d) are two graph segments based on the RSM. Assume that neither \( x \) nor \( y \) is a source. In Figure 4(c), there is no edge from \( y \) to \( x \), and the only incoming edge of \( y \) starts at \( x \), so 1 is satisfied. And for 2, we only need to consider Rule [y-x]. It can be observed that \( Nr(L_{xy}) = \{\langle b, n_0 \rangle\} \), \( L_{xy} = \{\ell_1\} \) and \( L_{yx} = \{\ell_2\} \). We have \( \langle b, n_0 \rangle \xrightarrow{\ell_{xy}=\ell_1} \langle b, n_1 \rangle \in R \) and \( \langle b, n_0 \rangle \xrightarrow{\ell_{yx}=\ell_2} \langle b, n_1 \rangle \). Thus, \((x, y)\) in Figure 4(c) is identified as foldable. In Figure 4(d), \( Nr(L_{xy}) = \{\langle b, n_0 \rangle\} \), \( L_{xy} = \{\ell_1\} \), \( L_{yx} = \{\ell_2\} \) and \( L_{yx} = \{\ell_2\} \). We have \( \langle b, n_0 \rangle \xrightarrow{\ell_{xy}=\ell_1} \langle b, n_1 \rangle \xrightarrow{\ell_{yx}=\ell_2} \langle b, n_0 \rangle \in R \) and \( \langle b, n_0 \rangle \notin \rangle \langle b, n_0 \rangle \). This means that Rule [x-x] of 2 is not satisfied. Thus, \((x, y)\) in Figure 4(d) is identified as not foldable.

The soundness of our graph folding principle manifests in that folding any node pair preserves reachability equivalence. Besides, since the states of \( Nr(L_{xy}) \) in Rule [x-y] are doubly checked by Rules [x-x] and [y-x], \( Nr(L_{xy}) \) in Rule [x-y] can be replaced by \( Nr(L_{yx}) \) for simplicity. Similarly, in Rule [y-x], \( Nr(L_y) \) can be replaced by \( Nr(L_{xy}) \). The principle decides foldability via only local information (i.e., incoming and outgoing edges of a node pair in graph \( G \)). Therefore, it does not exhaustively detect all foldable node pairs. For example, in Figure 1(b), if we only consider \( n_0 \) as the source and \( n_7 \) as the sink, folding \((v_1, v_2)\) does not affect CFL-reachability result, whereas we do not consider it as foldable because it violates Rule [x-y] in Figure 5.
4.3 Correctness of Folding Principle

We demonstrate the correctness of our graph folding by showing that Principles ① and ② in Theorem 4.1 satisfy Cond. 1 and Cond. 2 in Definition 4.2, respectively.

Lemma 4.1. Principle ① in Theorem 4.1 implies the exclusiveness condition (Cond. 1) of Definition 4.2.

Proof. A source-sink path in $G'$ that does not have an $xy$-FEQ class in $G$ can only be introduced when there is no edge from $y$ to $x$, as shown in Figure 6. In this case, when $y \in V_{src}$ or $y$ has an incoming edge from a node other than $x$, folding $(x, y)$ may introduce new source-sink paths from $y$ or the predecessors of $y$ to $x$ or the successors of $x$. Such new paths do not have any $xy$-FEQ class in $G$ and can lead to spurious reachable pairs. Principle ① avoids such incorrect foldings. □

Lemma 4.2. Principle ② in Theorem 4.1 implies the exclusiveness condition (Cond. 2) of Definition 4.2.

We first illustrate the objectives to which the four rules take effect. Figure 7 categorizes all paths containing at most one $xy$-subpath into four basic types of $xy$-FEQ classes, and shows the corresponding rules for ensuring consistency for each type. Briefly, Rule [x-y] ensures the consistency of the corresponding transition chains of paths where the $xy$-subpath starts and ends both with $x$ (Type 1). Rule [x-x] and Rule [x-y] together ensure the consistency of the corresponding transition chains of paths where the $xy$-subpath starts with $x$ and ends with $y$ (Type 2). Correspondingly, [y-y] and [y-x] ensure the consistency of Type 3, where $xy$-subpath starts and ends both with $y$, and Type 4, where $xy$-subpath starts with $y$ and ends with $x$.

Principle ② preserves the consistency for the four basic types based on Property 4.1 and Property 4.2, among which Property 4.1 is inferred from Property 3.1, and it indicates that $B^a \times \text{Nr}(L_x)$ and $B^a \times \text{Nr}(L_y)$ where $a = 0, 1, 2$ is enough to cover all cases (states) in $Q_x$ and $Q_y$. Property 4.2 guarantees the consistency of the endpoints of transition chains corresponding to the four basic types of $xy$-FEQ classes, which further ensures the satisfaction of Cond. 2. The detailed proofs of Property 4.1 and Property 4.2 are provided in our supplementary material.

Property 4.1. If the four rules in Figure 5 hold when $a \leq 2$, they also hold for all $a > 2$.

Property 4.2. With $P_{xy}$ denoting an $xy$-FEQ class belonging to the four basic types of Figure 7 and $p_{G'}$ denoting the $xy$-folded path of $P_{xy}$, if the four rules in Figure 5 hold, then (i) $p_{G'}$ is a reachable path iff $P_{xy}$ contains a reachable path, and (ii) when the paths of $P_{xy}$ do not end with $x$ or $y$, $p_{G'}$ corresponds to a sub-transition chain from $s_0$ to $s_k$ iff $P_{xy}$ also contains a path corresponding to a sub-transition chain from $s_0$ to $s_k$.

Proof of Lemma 4.2. We use the four basic types and the two properties to prove Lemma 4.2. In the original graph $G$, any path $p_G$ can be seen as the concatenation of subpaths $p_{G_1}p_{G_2} \cdots p_{G_k}$ such that (1) for each $i \in \{1, \cdots, k\}$, $p_{G_i}$ is either a path belonging to the four basic types of Figure 7 or a path not containing any $xy$-subpath, and (2) for all $i < k$, $p_{G_i}$ does not end with either $x$.

---

1Given two nodes $x$ and $y$ in a graph, an $xy$-path is a path comprised of edges joining $x$ and $y$. 

or $y$. Discussing $k = 1$ is trivial as it either belongs to the four basic types, which is covered by Property 4.2, or is a path not containing any $xy$-subpath, which is never changed by folding $(x, y)$.

Next, we consider $k > 1$ and start from $p_{G1}$. For the case that $p_{G1}$ does not belong to the four basic types, the corresponding transition chain is never changed by folding $(x, y)$, i.e., the corresponding states of nodes are not changed. For the case that $p_{G1}$ belongs to $P_{xy}$, one of the four basic types in Figure 7, according to Definition 4.1, the $xy$-folded path $p_{G1'}$ is exactly $P_{xy}$. Property 4.2 ensures that $p_{G1'}$ corresponds to a transition chain from $s_{init}$ to $s_1$ iff $P_{xy}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s_1$.

Analogously, for the concatenation $p_{G1}p_{G2} \cdots p_{Gk-1}$ that belongs to $P_{xyk-1}$, Property 4.2 ensures that the $xy$-folded path of $p_{G1}p_{G2} \cdots p_{Gk-1}$ (i.e., of $P_{xyk-1}$) corresponds to a transition chain from $s_{init}$ to $s_{k-1}$ iff $P_{xyk-1}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s_{k-1}$.

Finally, we consider the whole path $p_G$, which belongs to $P_{xy}$, and is folded into $p_{G'}$. (1) For the case that $p_{Gk}$ does not end with $x$ nor $y$, it can be inferred that $p_{G'}$ corresponds to a transition chain from $s_{init}$ to $s_k$ iff $P_{xy}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s_k$. (2) For the case that $p_{Gk}$ ends with $x$ or $y$, Property 4.2 also ensures that $p_{G'}$ corresponds to a transition chain from $s_{init}$ to $s_k$ such that $s_k \in F$ iff $P_{xy}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s'_k$ such that $s'_k \in F$. Therefore, for any $xy$-FEQ class $P_{xy}$ in $G$ and its $xy$-folded path $p_{G'}$ in $G'$, the four rules of Figure 5 ensures that $p_{G'}$ is a reachable path iff $P_{xy}$ contains a reachable path, which indicates the satisfaction of Cond. 2.

Putting Lemmas 4.1, 4.2 and Definition 4.2 together, folding a node pair $(x, y)$ satisfying ⋄ and ⋄ in Theorem 4.1 preserves reachability equivalence. Namely, Theorem 4.1 is correct.

## 5 GRAPH-FOLDING ALGORITHM

We then give an efficient graph-folding algorithm $Gr$ that traverses and folds the input graphs for CFL-reachability. Our algorithm implements the graph folding principle (Theorem 4.1). $Gr$ has a linear time complexity with respect to the number of nodes in input graphs.

### 5.1 Identifying Foldable Node Pairs

#### 5.1.1 Implementation of Graph Folding Principle

Algorithm 1 describes a practical realization of the folding principle in Theorem 4.1. The characteristic is that it only needs to compute the transitions involving the incoming and outgoing edges of a pair of adjacent nodes $x$ and $y$. Specifically, in lines 2–3, we set the values of $Q_x$ and $Q_y$ as their overapproximate supersets according to Principle...
Algorithm 1: Implementation of Graph Folding Principle

Function Identify(x, y)
1. $Q_x := \text{Nr} (L_{x}) \cup (B \times \text{Nr} (L_{x})) \cup (B^2 \times \text{Nr} (L_{x}))$
2. $Q_y := \text{Nr} (L_{y}) \cup (B \times \text{Nr} (L_{y})) \cup (B^2 \times \text{Nr} (L_{y}))$
3. if $L_{xy} = \emptyset$ and $(y \in V_{rc} \lor L_{yx} \neq \emptyset)$ then /* Theorem 4.1: ➀ */ return false;
4. if Check(x, y) and Check(y, x) then return true;
5. return false;
6. Procedure Check($o_1, o_2$)
7. for each $s_{o_1} \in Q_{o_1}$ do
8. for each $t_{o_2} \in L_{o_1,o_2}$ do
9. if not exists $(s_{o_1} \xrightarrow{t} s_{o_2} \in R \ s.t. \ s_{o_1} \xrightarrow{L_{o_1,o_2}} s_{o_2})$ then /* Rules [x-y] and [y-x] of Figure 5 */ return false;
10. for each $t_{o_2} \in L_{o_2,o_1}$ do
11. if exists $(s_{o_2} \xrightarrow{t} s_{o_1} \in R \ s.t. \ \neg(s_{o_1} \xrightarrow{L_{o_2,o_1}} s_{o_1}))$ then /* Rules [x-x] and [y-y] of Figure 5 */ return false;
12. return true;

② in Theorem 4.1, lines 4–5 and lines 6–7 verify the foldability of the input node pair based on Principles ➀ and ②, respectively.

Time Complexity. Algorithm 1 has a time complexity of $O(|B|^2 \times |N| \times |\Sigma|^3)$, where $N = \cup_{i \in \{1, \ldots, t\}} N_i$ is the collection of all local states of the RSM. In Algorithm 1, the time complexity of lines 2–5 can be regarded as $O(1)$ as they can be considered as lookups of hash tables. Then, the time complexity of Algorithm 1 depends on the subprocedure Check. We can first assume that a state transition $s_i \xrightarrow{t} s_j \in R$ can be performed in $O(1)$ time. According to Definition 4.5, given a set of label $L$ and two states $s_1$ and $s_2$, checking $s_1 \xrightarrow{L} s_2$ and $s_1 \xrightarrow{L} s_2$ needs to perform $O(|L|)$ transitions. Thus, the loops in lines 10–16 cost $(|Q_{o_1}| \times |L_{o_1,o_2}| \times |L_{o_2,y}| \times |Q_{o_1}| \times |L_{o_1,o_2}| \times |L_{o_2,y}| \times |L_{o_1,y}|)$, where the value of $Q_{o_1}$ is given in line 2 or line 3. We can find that $O(|Q_{o_1}|) = O(|B|^2 \times |N|)$, $|L_{o_1,o_2}| \leq |\Sigma|$, $|L_{o_2,y}| \leq |\Sigma|$, $|L_{o_1,o_2}| \leq |\Sigma|$ and $|L_{o_1,y}| \leq |\Sigma|$. Therefore, the time complexity of Algorithm 1 is $O(|B|^2 \times |N| \times |\Sigma|^3)$.

5.1.2 Efficient Identification for Foldable Node Pairs. Algorithm 1 runs fast for small RSMs, but the overall runtime increases when the RSMs become more complex. For real-world problems, we need a more efficient identification strategy. In fact, real-world CFL-reachability problems usually have an important trait—many node pairs share the same “pattern” of incoming and outgoing edges. Specifically, we define the “pattern” of a node pair $(x, y)$ as a tuple:

$$
\text{Pattern}_{(x, y)} = \langle \text{isSrc}(x), \text{isSrc}(y), L_{gx}, L_{gy}, L_{xy}, L_{yx}, L_{xg}, L_{yg} \rangle
$$

(3)

where isSrc(x) and isSrc(y) are two boolean variables denoting whether $x \in V_{rc}$ and whether $y \in V_{rc}$. The value of Pattern$((x, y))$ can be predefined or determined by consulting x and y and their incoming and outgoing edges. Obviously, for two node pairs $(x_1, y_1)$ and $(x_2, y_2)$, checking whether Pattern$((x_1, y_1)) = \text{Pattern}_{(x_2, y_2)}$ is much faster than calling Algorithm 1 twice.
Algorithm 2: Efficient Identification of Foldable Node Pairs

Ω⊕: a set holding the patterns of foldable node pairs.
Ω⊖: a set holding the patterns of non-foldable node pairs.

Function IsFoldable(x, y)
1. consult the incoming and outgoing edges to determine Pattern(x,y);
2. if Pattern(x,y) ∈ Ω⊕ then return true;
3. if Pattern(x,y) ∈ Ω⊖ then return false;
4. if Identify(x, y) then /* Algorithm 1 */
5. Ω⊕ := Ω⊕ ∪ {Pattern(x,y)}
6. return true;
7. Ω⊖ := Ω⊖ ∪ {Pattern(x,y)}
8. return false;

For the node pairs sharing the same pattern Pattern(x,y), we do not need to repeatedly invoke Algorithm 1 to check whether they are foldable. We use a set denoted by Ω⊕ to collect the patterns of node pairs that are already identified as foldable by Algorithm 1. When Ω⊕ is fully filled, we can identify whether a node pair is foldable by checking whether the pattern of the node pair is already in Ω⊕. Ω⊕ can be filled by verifying each possible pattern through Algorithm 1, where the number of invocation depends only on the size of the alphabet Σ. Besides, we provide another strategy in Algorithm 2, which dynamically constructs Ω⊕ during the process of graph folding with a set Ω⊖ holding patterns of non-foldable node pairs.

5.2 Overall Algorithm

Algorithm 3 describes the overall graph-folding algorithm Gf. In addition to Ω⊕ and Ω⊖ in Algorithm 2, Algorithm 3 maintains two sets Vvisited and Evisited to collect the visited nodes and edges. The procedures Gf and Visit scan the input graph via a depth-first traversal. When visiting a node x, the algorithm calls the identification procedure IsFoldable for each unvisited direct successor y of x to check whether (x, y) is foldable. If (x, y) is foldable, the algorithm uses Fold(x, y) in lines 13–17 to fold (x, y) and update representative nodes.

Complexity. Algorithm 3 has a linear time complexity with respect to the number of nodes in the input graph. For real-world problems where the RSM is far smaller than the graph, the cost for identifying and folding a node pair (lines 10–11) can be considered as O(1) time. The ordinary depth-first traversal costs O(|V| + |E|) time. However, different from the ordinary depth-first traversal, line 9 ensures that each node should not be visited twice through different edges. Lines 8–12 show that the number of method invocations for IsFoldable(x, y) and Fold(x, y) does not exceed the number of visited nodes. In Algorithm 3, each node is visited once, hence the time complexity of Algorithm 3 is O(k|V|) where k is a constant representing the time for identifying and folding a node pair. Namely, Gf has a linear time complexity with respect to the number of nodes of the input graph.

6 EXPERIMENTS

We evaluate our graph-folding algorithm Gf by applying it to two popular client analyses for C/C++: value-flow analysis [Sui et al. 2014] and alias analysis [Zheng and Rugina 2008]. In particular, we study the performance of Gf from two aspects: (1) the performance of Gf in reducing the input graph sizes and (2) the speedups and memory overhead reductions for CFL-reachability, with the input graphs simplified by Gf.
Algorithm 3: Graph-Folding Algorithm $GF$

$V_{visited}$: a set holding visited nodes.
$E_{visited}$: a set holding visited edges.

Function $GF(G, R)$

1. set $\Omega_\emptyset, \Omega_\emptyset, V_{visited}$ and $E_{visited}$ as $\emptyset$;
2. for each $x \in V$ do
3.     if $x \notin V_{visited}$ then Visit($x$);
4. return $G$;

Procedure Visit($x$)

5. $V_{visited} := V_{visited} \cup \{x\}$
6. for each $x \rightarrow y \in E$ s.t. $y \notin V_{visited}$ and $x \rightarrow y \notin E_{visited}$ do
7.     add all edges joining $x$ and $y$ into $E_{visited}$;
8. if IsFoldable($x, y$) then
9.     Fold($x, y$);
10. Visit($y$)

Procedure Fold($x, y$)

11. Remove all edges joining $x$ and $y$;
12. for each $z \in V$ s.t. $Rep(z) = y$ do
13.     $Rep(z) := x$; /* Update representative nodes */
14. Remove node $y$; /* Merge $y$ into $x$ */

In summary, the experimental results are promising. On average, by reducing 60.96% nodes and 42.67% edges of the input graphs, $GF$ accelerates context-sensitive value-flow analysis by $4.65 \times$ with a memory reduction rate of 57.35%; by reducing 38.93% nodes and 35.61% edges, $GF$ accelerates field-sensitive alias analysis by $3.21 \times$ with a memory reduction rate of 65.19%. $GF$ is also well-compatible with existing techniques. The combination of $GF$ with SCC [Nuutila and Soisalon-Soininen 1994] and INTERDYCK [Li et al. 2020] reduces up to 72.26% and 58.85% edges from the input graph of context-sensitive value-flow analysis and field-sensitive alias analysis, respectively, accelerating the analyses by at most $9.70 \times$ and $6.30 \times$.

6.1 Experimental Setup

We have implemented $GF$ on top of LLVM-12.0.0, and conducted our experiments on a platform consisting of an eight-core 2.60 GHz Intel Xeon CPU with 128 GB memory, running Ubuntu 18.04. In our experiments, the patterns of foldable node pairs, i.e., $\Omega_\emptyset$ in Section 5.1.2, are precomputed.

Grammar Selection. We evaluate graph folding based on two client analyses, a context-sensitive value-flow analysis [Sui et al. 2014] and a field-sensitive alias analysis [Zheng and Rugina 2008]. Figure 8 gives the corresponding context-free grammars. Both grammars extend Dyck languages with regular languages, which are representative of many CFL-reachability-based program analyses. In particular, Dyck languages have been extensively used to model function calls/returns, references/dereferences, and field writes/reads [Kodumal and Aiken 2004]. Regular languages have been used to model control/value flows. Moreover, the two client analyses have also been evaluated in existing work [Lei et al. 2022; Wang et al. 2017]. Figure 9 gives the corresponding RSM representations.

Value-flow Analysis. We conduct context-sensitive value-flow analyses on sparse value-flow graphs (SVFGs) [Sui et al. 2014]. The RSM for context-sensitive value-flow analysis is displayed in
Recursive State Machine Guided Graph Folding for Context-Free Language Reachability

\[ S ::= \text{call}_i S \text{ret}_i \mid S S \mid a^* \]

\[ M ::= \overline{A} V d \]

\[ V ::= (M? \overline{A})^* V (a M?)^* \mid \overline{f}_i V f_i \mid M? \]

(a) Value-flow analysis.

(b) Alias analysis.

Fig. 8. CFGs for context-sensitive value-flow analysis and field-sensitive alias analysis.

\[ \begin{align*}
M_1 & : \langle n_1 \rangle \\
\text{call} & \rightarrow n_1 \\
\text{ret}_i & \rightarrow b_1 : M_1
\end{align*} \]

(a) Value-flow analysis.

\[ \begin{align*}
M_1 & : \langle n_1 \rangle \\
\overline{f}_i & \rightarrow \langle n_1 \rangle \\
\overline{d} & \rightarrow \langle n_2 \rangle \\
\overline{f}_i & \rightarrow \langle n_3 \rangle \\
\overline{d} & \rightarrow \langle n_4 \rangle
\end{align*} \]

(b) Alias analysis. The doubly-circled nodes denote the exits of \( M_1 \).

Fig. 9. RSMs for context-sensitive value-flow analysis and field-sensitive alias analysis.

Figure 9(a), where “call\(_i\)” and “ret\(_i\)” denote call and return of a callsite, whose index is \( i \); and “\( a \)” denotes an assignment instruction. In the RSM, the initial state and the final state are both \( \langle n_1 \rangle \), and the unique box \( b \) has a subscript \( i \), matching the index of call\(_i\) and ret\(_i\). In this reachability problem, nodes denoting the allocation/deallocation sites are marked as sources/sinks. It is worthwhile to point out that although the RSM in Figure 9(a) only focuses on context-sensitivity, the analysis is field-sensitive because each field object in SVFGs is already represented as a distinct node.

**Alias Analysis.** Figure 9(b) gives the RSM for all-pair field-sensitive alias analyses, where all nodes are considered as both sources and sinks. In the RSM, \( a \) denotes an assignment, \( d \) denotes a pointer dereference, and \( f_i \) denotes the address of the \( i \)-th field. The RSM contains three boxes \( b_1, b_2, \) and \( b_3 \), which are all mapped to \( M_1 \). \( b_3 \) has a subscript \( i \) matching the field index of \( f_i \) and \( \overline{d} \). The RSM has an initial state \( \langle n_1 \rangle \) and four accepting states \( \langle n_1 \rangle, \langle n_2 \rangle, \langle n_3 \rangle \) and \( \langle n_4 \rangle \). The alias analysis is conducted on program expression graphs (PEGs), which are bidirected, i.e., for each \( v_i \xrightarrow{\text{L}} v_j \in E \) where \( t \in \Sigma \), there is a reverse \( v_j \xrightarrow{\text{R}} v_i \in E \).

**Benchmarks.** We selected ten popular GitHub open-source C/C++ programs to benchmark our analysis in Table 2. We chose these programs because they are diverse in terms of functionalities as they include: development (astyle, nvim), version control (git-checkout), compiler (janet, mruby), database (psql, redis\_ccli), computer vision (opencv\_test\_video), window manager (i3) and terminal multiplexer (tmux). The SVFG and PEG of each program are generated by the open-source tool SVF [Sui and Xue 2016b] from the bitcode files compiled by Clang-12.0.0 with the -O3 flag and integrated by Whole Program LLVM.

**Evaluated Algorithms.** Our graph-folding algorithm \( Gf \) is a preprocessing technique for speeding up CFL-reachability. Therefore, in our evaluation, we need to choose a baseline CFL-reachability algorithm (for solving CFL-reachability). To this end, we select two recent CFL-reachability solvers, Graspan [Wang et al. 2017] and POCR [Lei et al. 2022]. In general, POCR runs faster than Graspan. Therefore, we pick POCR [Lei et al. 2022] as the baseline CFL-reachability solver. However, POCR ran out of memory on two benchmark programs git\_checkout and mruby. For both programs,
Table 2. Benchmark info and results of the baseline CFL-reachability solver. #Node and #Edge denote the numbers of nodes and edges of each input graph, respectively. P-Edge% denotes the percentage of parenthesis edges out of the total edges of each input graph. Time/s and Mem./GB denote the runtime and memory overhead of the baseline for analyzing each program, measured in seconds and gigabytes, respectively.

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Value-flow analysis</th>
<th>Alias analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Node</td>
<td>#Edge</td>
</tr>
<tr>
<td>1.astyle</td>
<td>227140</td>
<td>294939</td>
</tr>
<tr>
<td>2.git_checkout</td>
<td>488092</td>
<td>919390</td>
</tr>
<tr>
<td>3.i3</td>
<td>145259</td>
<td>212761</td>
</tr>
<tr>
<td>4.janet</td>
<td>227081</td>
<td>359552</td>
</tr>
<tr>
<td>5.mruby</td>
<td>226528</td>
<td>340374</td>
</tr>
<tr>
<td>6.nvim</td>
<td>332733</td>
<td>402753</td>
</tr>
<tr>
<td>7.opencv_test_video</td>
<td>385060</td>
<td>509439</td>
</tr>
<tr>
<td>8.psql</td>
<td>157014</td>
<td>228604</td>
</tr>
<tr>
<td>9.redis-cli</td>
<td>231372</td>
<td>367291</td>
</tr>
<tr>
<td>10.tmux</td>
<td>243828</td>
<td>390084</td>
</tr>
</tbody>
</table>

* The results marked by * are obtained by Graspan [Wang et al. 2017] because POCR ran out of memory.

we used Graspan to obtain the results. Table 2 displays the graph statistics and the results of the baseline approach. We compare our graph-folding algorithm Gf against two existing preprocessing techniques: cycle elimination (SCC) [Nuutila and Soisalon-Soininen 1994; Tarjan 1972] and graph simplification for interleaved Dyck-reachability (INTERDYCK) [Li et al. 2020]. For each input graph, SCC detects and collapses cycles comprised of \(-\)-edges, and INTERDYCK detects and eliminates the non-Dyck-contributing parenthesis edges. Note that, unlike Gf, neither SCC nor INTERDYCK is a general technique for CFL-reachability.

Evaluation of Correctness. To ensure correctness, we check whether the baseline CFL-reachability solver can compute equivalent solutions from the original input graph \(G\) and the simplified version \(G'\), which is folded from \(G\) by Gf. Specifically, for each program, after obtaining the solutions from \(G\) and \(G'\), we expand all reachable pairs comprised of representative nodes \((v_i, v_j)\) in \(G'\) into \(\{(v_i, v_j) \mid \text{Rep}(v_i) = v'_i, \text{Rep}(v_j) = v'_j\}\), and comparing the reachable pairs with the solutions on \(G\). Our comparison demonstrates that the expanded solutions on the folded graphs are identical to the solutions on the original graphs.

6.2 Performance in Reducing Graph Sizes

Figures 10 and 11 depict the reduction rates of nodes and edges in the input graphs of value-flow analysis and alias analysis, respectively. As INTERDYCK does not remove nodes from a graph, the node reduction charts (Figure 10(a) and Figure 11(a)) do not include the information of INTERDYCK. A comparison of SCC, INTERDYCK and Gf shows that Gf is more efficient than SCC and INTERDYCK in reducing the size of the input graphs for both clients. Specifically, by comparing Gf with SCC, we find that cycle elimination reduces only a small number of nodes and edges in the preprocessing stage. Prior work shows that cycle elimination works well in constraint-based pointer analysis because the edges typically denote subset constraints [Hardekopf and Lin 2007a; Pereira and Berlin 2009]. However, in CFL-reachability, the edges in the input graph are labeled with different letters. By comparing Gf with INTERDYCK, we find that INTERDYCK can reduce the number of parenthesis edges. But because the proportions taken by parenthesis edges are not large (see Columns 4 and 9 of Table 2) in these two real-world clients, whereby the effectiveness of INTERDYCK is limited.
Besides, a comparison between node reduction rates and edge reduction rates of GF shows that folding tends to increase the density (#Edge/#Node) of the graph.

We also study the combinations GF+SCC and GF+SCC+INTERDYCK, which are depicted as the dashed lines in Figures 10 and 11. The result shows that GF complements SCC and INTERDYCK well, i.e., they can be used together to further improve the performance of CFL-reachability. Table 3 shows the runtime of GF, SCC and INTERDYCK when they are applied separately to the graphs. The results show that SCC is the fastest, followed by GF, and INTERDYCK is much slower. This is because implementing INTERDYCK in these two clients needs to (1) detect and contract all non-parenthesis edges, (2) use the FastDyck algorithm [Zhang et al. 2013] to find Dyck-contributing edges and mark the “anchor” nodes and (3) detect and remove non-Dyck-contributing edges, where (1) costs $O(|V|^2)$ time, (2) costs $O(|V| + |E|\log|E|)$ time and (3) costs $O(|E|\log|E|)$ time.

### 6.3 Speedup and Memory Overhead

Figure 12 shows the speedups of CFL-reachability by GF, SCC, INTERDYCK, and their combinations. Taking the edge reduction rate (Figure 10(b) and Figure 11(b)) into consideration and comparing the speedups among different graph simplification approaches, we observe that the reduction of more edges from the graphs helps to improve CFL-reachability solving. Comparing the speedups of a graph simplification approach among different programs, we can see that larger edge reduction rates usually, but not always, result in larger speedups. This is because the runtime of CFL-reachability solving depends not only on the size of the input graph but also on other graph traits, such as density and edge types.
Table 3. Runtime of Gr, SCC and InterDyck, measured in seconds.

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Value-flow analysis</th>
<th></th>
<th>Alias analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gr</td>
<td>SCC</td>
<td>InterDyck</td>
</tr>
<tr>
<td>1.astyle</td>
<td>3.70</td>
<td>0.69</td>
<td>273.04</td>
</tr>
<tr>
<td>2.git_checkout</td>
<td>15.29</td>
<td>2.09</td>
<td>1625.84</td>
</tr>
<tr>
<td>3.i3</td>
<td>2.63</td>
<td>0.27</td>
<td>177.74</td>
</tr>
<tr>
<td>4.janet</td>
<td>4.01</td>
<td>0.80</td>
<td>464.37</td>
</tr>
<tr>
<td>5.mruby</td>
<td>15.74</td>
<td>0.82</td>
<td>718.04</td>
</tr>
<tr>
<td>6.nvim</td>
<td>16.72</td>
<td>0.94</td>
<td>152.02</td>
</tr>
<tr>
<td>7.opencv_test_video</td>
<td>57.80</td>
<td>2.35</td>
<td>541.36</td>
</tr>
<tr>
<td>8.ps dol</td>
<td>2.39</td>
<td>0.30</td>
<td>130.31</td>
</tr>
<tr>
<td>9.redis-cli</td>
<td>7.34</td>
<td>0.64</td>
<td>143.18</td>
</tr>
<tr>
<td>10.tmux</td>
<td>10.13</td>
<td>0.75</td>
<td>1040.18</td>
</tr>
</tbody>
</table>

Fig. 12. Speedups of CFL-reachability by Gr, SCC, InterDyck and their combinations.

Figure 13 shows the reduction rates of memory overheads by Gr in the two client analyses. Comparing the performance of git_checkout and mruby (solved by Graspan) with other benchmarks (solved by POCR) in Figure 13(a), we can see that the memory overhead reduction by Gr for POCR is much more significant than for Graspan. This is also observable in Figure 13(b), where all benchmarks are solved by POCR. The insight is that when solving CFL-reachability, POCR maintains a spanning tree for each node to hold the “transitive” edges (i.e., a-edges) for the graphs. Reducing the nodes and edges in the graph also reduces the auxiliary spanning trees, leading to those dramatic reduction rates of memory overhead. A comparison of Figure 12 and Figure 13 indicates that, in general, larger speedups correspond to larger memory overhead reductions. What lies behind this correlation is that solving CFL-reachability is adding new edges to the graph according to existing edges. Therefore, a larger speedup means a larger reduction rate of edges that need to be created and stored. This ultimately reduces the memory overhead.
6.4 Discussions

Graph Folding and Domain-Specific Edge Contraction. Perhaps the best-known domain-specific edge contraction technique in static analysis is the offline variable substitution (OVS) for pointer analysis [Rountev and Chandra 2000]. Specifically, OVS can contract a copy-edge from a node \( u \) to another node \( y \) if \( y \) does not have its address taken and \( y \) does not have any other incoming copy-edge. In our evaluated alias analyses, \( G_f \) folds a node pair \((x, y)\) where there is an \( a \)-edge from \( x \) to \( y \) if \( y \) does not have any incoming \( d \)-edge and \( f_i \)-edge, and does not have any incoming \( a \)-edge not from \( x \). The intrinsic meaning of the folding condition of \( G_f \) is equivalent to that of OVS.\(^3\) Thus, OVS can be viewed as an instantiation of graph folding in pointer/alias analysis.

Foldability of Node Pairs v.s. Self-Loops in the RSM. By observing the RSMs of the two clients of our experiments (Figures 9(a) and 9(b)), a common feature is that both RSMs contain self-loops, e.g., \( n_1 \xrightarrow{a} n_1 \) in the RSM of Figure 9(a). It is interesting to note that a foldable node pair does not necessarily need to be joined by an edge that corresponds to a self-loop in the RSMs. Consider the node pair \((x, y)\) in Figure 4(c), whose foldability is discussed in Example 4.3. In the example, the edge \( x \xrightarrow{f_i} y \) does not correspond to a self-loop in the RSM of Figure 4(a). In fact, the RSM in this example does not contain any self-loop.

Choosing Sources and Sinks. To further demonstrate the effectiveness of our approach, we perform an additional double-free analysis by choosing source-sink pairs different from the memory leak detection application. The double-free problem on SVFGs can also be seen as a source-sink reachability problem by reusing the grammar in Figure 8(a) and the RSM in Figure 9(a). A simplified version of the analysis requires that each allocated memory must be freed at most once when traversing SVFGs. Specifically, any deallocation site of variable \( p \) should never reach any other deallocation sites of variable \( q \), where \( p \) and \( q \) are aliases. As a result, we choose deallocation sites as both the source and sink in our simplified double-free analysis. Figure 14 gives the number of deallocation sites and the reduction rates of nodes and edges by \( G_f \) for each benchmark. We can see that \( G_f \) can achieve similar reduction rates with a different set of sources and sinks. Compared with the original value-flow analysis (Figure 10), the reduction rates in Figure 14 are slightly lower, and the difference is roughly the number of free sites. This is because the sources in the simplified double-free analysis usually have incoming \( a \)-edges. The node pairs involving such edges cannot be folded by \( G_f \), but are foldable in the value-flow analysis for memory leak detection.

7 RELATED WORK

Due to the (sub)cubic time complexity of CFL-reachability with respect to the input graph size, improving the practical performance of CFL-reachability is desirable. However, the truly subcubic

\(^3\)In particular, an \( f_i \)-edge is regarded as a copy-edge with an offset \( i \) in field-sensitive analysis [Pearce et al. 2007].
CFL-reachability algorithms [Le Gall 2014; Williams and Williams 2018] are impractical due to the large constant. In the literature, significant progress has been made only for specific clients or particular context-free grammars [Bastani et al. 2015; Chatterjee et al. 2018; Yan et al. 2011; Zhang et al. 2013, 2014; Zheng and Rugina 2008].

Due to the impracticability of reducing the time complexity of CFL-reachability, reducing the size of the input graph has been studied and adopted by many researchers. The most prevalent technique is cycle elimination [Nuutila and Soisalon-Soininen 1994; Tarjan 1972], which has been widely applied to an alternative form of CFL-reachability, i.e., constraint solving [Fähndrich et al. 1998; Hardekopf and Lin 2007a; Kodumal and Aiken 2004; Pereira and Berlin 2009; Xu et al. 2009]. What limits the applicability of cycle elimination in CFL-reachability is that whether a cycle can be merged depends on the context-free grammar. Edge contraction has also been studied for domain-specific clients [Hardekopf and Lin 2007b; Rountev and Chandra 2000]. Such methods can be seen as specializations of our graph folding in these clients. Recently, Li et al. [2020] proposed an algorithm to remove the non-Dyck-contributing edges for interleaved-Dyck-reachability problems. The approach only focuses on reducing redundant parenthesis edges, but there are also numerous non-parenthesis edges to be eliminated in real-world problems.

Reducing the redundancy of on-the-fly CFL-reachability solving is also studied by researchers [Bravenboer and Smaragdakis 2009; Jordan et al. 2016; Lei et al. 2022; Wang et al. 2017]. These techniques are orthogonal with our Gr, and they can be applied together to solve a CFL-reachability problem. CFL-reachability is also studied in two alternative forms called (1) recursive state machines (RSMs) [Alur et al. 2005b, 2006; Benerecetti et al. 2010; Chatterjee et al. 2015; Chaudhuri 2008], and (2) pushdown automata (PDA) [Chatterjee and Velner 2012; Reps et al. 2007, 2005; Rytter 1983; Späth et al. 2019; Wojciech and Rytter 1985]. Several approaches are proposed to simplify visibly pushdown automata (VPA) [Gauwin et al. 2019; Heizmann et al. 2017]. By exploiting the equivalence property of states, Heizmann et al. [2017] simplify visibly pushdown automata (VPA) to make a complex automaton simple. Prior work shows that the minimization of VPA is NP-complete [Gauwin et al. 2019]. These techniques are also orthogonal to our graph folding, as they focus on simplifying the CFL itself rather than the graph.

8 CONCLUSION

This paper has described a new graph folding technique for CFL-reachability. It has formulated the criteria for foldable node pairs and proposed a graph-folding algorithm Gr. Experimental results of context-sensitive value-flow analysis and field-sensitive alias analysis for C and C++ on ten open-source programs show substantial performance improvements enabled by Gr via reducing the graph sizes. On average, Gr reduces 60.96% of nodes and 42.67% of edges of the input graphs for value-flow analysis, obtaining a speedup of 4.65× and a memory usage reduction of 57.35%, and reduces 38.93% of nodes and 35.61% of edges of the input graphs for alias analysis, obtaining a speedup of 3.21× and a memory usage reduction of 65.19%.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers and the shepherd Peisen Yao for valuable feedback on earlier drafts of this paper, which helped improve its presentation. This research is supported by Australian Research Grants DP210101348 and FT220100391; by Amazon under an Amazon Research Award in automated reasoning; by the United States National Science Foundation (NSF) under grants No. 1917924 and No. 2114627; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

DATA AVAILABILITY STATEMENT

Materials for our evaluation are publicly available [Lei et al. 2023] and can be used to reproduce the data of our experiment. The code is sourced from the project https://github.com/kisslune/POCR.

REFERENCES


