Calling-to-Reference Context Translation via Constraint-Guided CFL-Reachability

Cheng Cai  
University of California, Irvine  
ccaie@uci.edu

Qirun Zhang  
University of California, Davis  
qrzhang@ucdavis.edu

Zhiquiang Zuo  
State Key Lab. for Novel Software Technology, Nanjing University

Khanh Nguyen  
University of California, Irvine  
khanhtn1@uci.edu

Guoqing Xu  
University of California, Irvine  
harry.g.xu@uci.edu

Zhendong Su  
University of California, Davis  
su@ucdavis.edu

Abstract

A calling context is an important piece of information used widely to help developers understand program executions (e.g., for debugging). While calling contexts offer useful control information, information regarding data involved in a bug (e.g., what data structure holds a leaking object), in many cases, can bring developers closer to the bug’s root cause. Such data information, often exhibited as heap reference paths, has already been needed by many tools.

The only way for a dynamic analysis to record complete reference paths is to perform heap dumping, which incurs huge runtime overhead and renders the analysis impractical. This paper presents a novel static analysis that can precisely infer, from a calling context of a method that contains a use (e.g., read or write) of an object, the heap reference paths leading to the object at the time the use occurs. Since calling context recording is much less expensive, our technique provides benefits for all dynamic techniques that need heap information, significantly reducing their overhead.

CCS Concepts → Software and its engineering → Automated static analysis; Dynamic analysis;

Keywords Static analysis, dynamic analysis, heap dump

1 Introduction

Context information is critical for program understanding and bug diagnosis. The current development practices, especially for object-oriented languages, decompose the functionality into small methods and the data into multi-layer structures. For any dynamic analysis that aims to find bugs or other runtime problems, it is extremely helpful to report not only important events (e.g., bugs) but also the contexts under which they occur. Two major kinds of contexts used in practice are calling contexts (i.e., abstractions of call stacks) and reference contexts (i.e., abstractions of heap reachability).

Calling contexts are commonly used in language implementations and dynamic analyses — for example, they are often reported together with exceptions or other types of events to help developers make better sense of the runtime stack when an event occurs. Calling contexts are relatively cheap to collect. The past decade has seen a proliferation of efficient calling context profiling techniques, including stack walking [34], context tree profiling [54], or context encoding [9, 19, 39] and decoding [7, 42]. For instance, probabilistic context encoding (PCE) [9] adds only 3% overhead to a JVM, making it deployable for production systems.

Often times postmortem diagnosis can be made much easier with the reference context of a bug. A reference context encodes the data information of the objects involved in the bug, revealing the logical connections among these objects. For instance, memory leak detectors typically report leaking objects only with their calling contexts [1, 8, 21, 25, 43, 44]. To understand why an object leaks, however, developers often need to trace heap accesses themselves to identify the problematic data structure that holds references to the leaking object. It is hard to develop a fix without understanding where these unnecessary references are located.

For race detectors [7, 11], the reference context for a racy object o identifies objects that directly and transitively reference o, immediately revealing the root cause if the race is due to synchronization bugs in o’s owning class — e.g., a synchronized Hashtable should be used in a multi-threaded environment as opposed to an unsynchronized HashMap. Moreover, commercial diagnosis tools such as IBM WAIT [15] — previously known as Yeti [23] — and YourKit [48] all need to
analyze references (derived from heap dumps) to provide insights to performance problems.

Obtaining reference contexts is notoriously difficult since objects do not carry backward pointers. Existing profiling techniques that piggyback on the garbage collector (GC) can only track forward references. They cannot report the reference paths leading to an object of interest. The only way to obtain the complete set of reference paths for the object is taking heap snapshots. While production JVMs provide functionality to perform heap dumping, writing content of a JVM heap of size several to dozens of gigabytes to disk results in extremely slow executions. This approach is also inefficient because a client usually needs only a very small portion of the heap (e.g., containing the reference paths leading to a number of interesting objects) for problem diagnosis.

We use a concrete example to compare the overhead between heap dumping and calling context profiling. Specifically, we consider calling context uptree (CCU) [14], which is a JVM-based technique for efficiently profiling calling contexts. The CCU implementation also contains a memory leak detector [8] and a data race detector [7]. Figure 1 shows an overhead comparison between the vanilla CCU that profiles calling contexts for each object (CCU-vanilla), CCU that dumps the whole heap (Heap-dump); all of the DaCapo [6] programs were run on a FastAdaptive build of Jikes RVM 3.1.1 configured with a generational MarkSweep GC; the heap size given for each program was twice as large as the minimum heap size required for the program.

Our Contributions. We propose a static analysis that can precisely infer the heap reference paths (reference contexts) of an event object from a call stack (calling context) captured by a dynamic analysis, thereby providing immediate benefit to all dynamic analyses and bug finding tools that require reference path information but could not obtain it due to the prohibitively high profiling cost. At the core of our technique is a translation framework that makes a static connection between call stacks and reference paths. Our work is inspired by recent advances in the context-free language (CFL) reachability formulation of object flow [37, 45, 50], which makes an interplay between method calls (i.e., represented by a balanced-parenthesis language $\mathbb{M}$) and heap accesses (i.e., represented by a balanced-bracket language $\mathbb{H}$).

Our translation framework extends the languages $\mathbb{M}$ and $\mathbb{H}$, respectively, to $\overline{\mathbb{M}}$ and $\overline{\mathbb{H}}$, allowing them to contain unbalanced parentheses and brackets under certain circumstances. Given a calling context $c$, an object $o$ of interest recorded by a dynamic analysis, and a variable $v$ that points to object $o$, our static analysis computes a set of potential reference chains that end at $o$ at the time $c$ is recorded. We do so by solving $\overline{\mathbb{M}} \cap \overline{\mathbb{H}}$-reachability from $v$ to each object $o'$, which directly or transitively references $o$, such that (1) the string of the unbalanced method entries/exports in language $\overline{\mathbb{M}}$ on each path from $v$ to $o'$ satisfies a prefix constraint w.r.t. the collected calling context $c$ and (2) the string of the unbalanced field accesses in $\overline{\mathbb{H}}$ on the same path satisfies the unbalanced constraint w.r.t. a user-specified number $n$ of unbalanced brackets. These $n$ unbalanced brackets of field accesses can be used to derive reference paths of length up to $n$ for $o$.

We refer to our formulation as constraint-guided (CG) CFL-reachability. We have implemented this technique in Soot [35] and conducted extensive experiments with a real leak detector [8] and data race detector [7] implemented in CCU [14] on the DaCapo benchmark set [6]. For the DaCapo programs, our translation takes, on average, 13.7 seconds to compute reference contexts for each query, while executing a program with a heap dump typically can take several hours. Moreover, our algorithm produces precise reference contexts compared against the actual reference paths captured by heap dumps — the statically inferred reference paths are only 2% more and 30% shorter than their dynamic counterparts.

Finally, we have manually investigated 20 leak reports and 20 data race reports for each program. With the help of the reported reference contexts, we were able to quickly find the root causes and develop fixes for most of them (Section 6.2).

2 Motivating Example

Figure 2 shows a real example that illustrates the importance of having reference paths in a leak report. The report is generated by a well-known leak detector Sleigh [8]. Sleigh tracks object staleness (i.e., time since an object’s last use). When the staleness of an object exceeds a threshold, Sleigh
void runCompare(ISelection s) {
    ResourceCompareInput input = new ResourceCompareInput(s); // o2
    openCompareEditorOnPage(input, fWorkbenchPage);
    useValue(input);
    ...  // never use input again
}

void openCompareEditorOnPage(CompareEditorInput in,
    IWorkbenchPage p) {

    NavigationHistory info = new NavigationHistory(...); // o2
    info.add(in); // input is added into an ArrayList in info
    Editor.addHistoryItem(info); // info is cached
}

void useValue(ResourceCompareInput in) {
    p = in.value;  // never use input again
}

class NavigationHistory {
    void add(Object obj) {
        Object[] tmp = this.array;
        tmp[... ] = obj;
    }
}

(a) Simplified code snippet.

(b) Interprocedural symbolic points-to graph for the program in (a); variables (name) and objects (o2) are named based on line numbers i while symbolic nodes Sj are named sequentially on integer j.

Figure 2. A real memory leak example from Eclipse that shows the importance of reporting heap reference paths; inverse edges in the SPG are omitted.

reports its last use site (i.e., a read or write) together with the calling context under which this use occurs.

The example is a significantly simplified version of the Eclipse compare plugin, which leaks memory when repeatedly comparing two zip files. Figure 2a shows the code snippet. Method runCompare() is the entry point of the plugin that takes as input the files selected by an Eclipse user. It first creates a ResourceCompareInput object to wrap the files and calls method openCompareEditorOnPage() to open the compare editor. After line 4, object o2 (created at line 2) is never used again, and it is reported as a memory leak. In this case, Sleigh reports that the last use site of object o2 is line 14, and the calling context for this use is runCompare() → useValue().

This calling context is not particularly useful because it does not explain where the leaking object is held and why the reference is not cleared. Furthermore, method openCompareEditorOnPage() is defined in a different (UI) plugin, which is used to create GUI gadgets for displaying comparison results. Object o2 flows into this plugin and gets cached there. However, following the reported calling context, the developer may focus her debugging effort on inspecting the code of plugin compare (as all the methods on the call chain are in compare), which would never bring her to the actual cause.

The root cause can be quickly uncovered if the following reference context is reported: o2 is appended to an array (accessed at line 18), which is, in turn, referenced by the NavigationHistory object, created at line 9 in the UI plugin. This reference path pinpoints the problematic references: the NavigationHistory object is the root that transitively holds the leaking object and, hence, the leak should be fixed by removing the object from NavigationHistory.

We can see that identifying the root cause of a runtime problem is often non-trivial with only calling context information available. Our translation framework overcomes this challenge by translating profiled calling contexts to reference contexts. Our translation algorithm operates on a graph representation of the program. While our formulation itself is independent of the underlying graph representation, we choose the symbolic points-to graph (SPG) [45, 46] to implement our algorithm for the reasons detailed in Section 5.1.

SPG provides a sound and precise approximation of the runtime object flow. Figure 2b depicts the SPG representation of our example. An SPG for a method is a locally resolved points-to graph where symbolic nodes are introduced to represent objects that are not visible in the method. The graph is bidirected as a prerequisite for the CFL-reachability-based pointer analysis [28, 37]. In Figure 2b, circles represent variables (named with the actual variable names followed by line numbers) and boxes represent (1) allocation sites (named with o plus line numbers) and (2) symbolic objects (named with S plus an integer that sequentially grows). Edges without annotations are points-to edges from variables to objects/symbolic nodes. Annotated edges represent calls/returns (annotated with call site IDs) or field points-to relationships (annotated with field names).

For example, node in13 corresponds to variable in defined at line 13, which is an alias of node input2. Edge o2 → S13 corresponds to the collected calling context that involves the call site at line 4. Symbolic node S13 is a placeholder of object o2 that variable in (node in13) points to at its last use site line 14. This symbolic node is introduced because SPGs are constructed intraprocedurally — o2 is not visible when the SPG for method openCompareEditorOnPage() is constructed. SPGs for different methods are connected trivially at call sites through formal-actual parameter pairing and return values to form an interprocedural SPG (ISPG).

ISPG is used in [46] to quickly answer on-demand alias queries. In particular, a memory-alias (MemAlias) relation is defined over the set of symbolic and object nodes — given two symbolic nodes or one symbolic and one object node, they are memory aliases if there exists a MemAlias-path on
the ISPG between them. This formulation has been used by many techniques [36, 46, 50] for answering alias queries. Section 5 discusses more details of the SPG and why it is particularly suitable for our algorithm.

Given a user-specified length parameter \( n = 2 \), our translation algorithm finds a path on the ISPG from node \( in_{13} \) (the variable accessed at the last use site) to node \( o_{9} \) (the container object), based on the last use statement at line 14 and the calling context \( call_{4} \) profiled by the memory leak detector. The sequence of the labels on this path is finally converted to node \( o_{2} \), where \( o_{array} \) is the allocation site (not shown in the figure) for the array accessed at line 19. This reference path immediately reveals that the problematic NavigationHistory object at line 9 is the root cause of the leak.

3 Preliminaries

This section defines the problem of static context translation. Here we consider a Java-like language while the technique can easily generalize to other object-oriented languages.

3.1 Calling Context and Reference Context

Our context translation takes as input (1) an event collected by a dynamic analysis that contains either a store or a load of a heap object and (2) the calling context under which the event occurs. While practical dynamic analysis may report a variety of information, most runtime problems (e.g., bugs or performance issues) have strong correlation with heap stores or loads. Moreover, stores and loads are the building blocks of high-level semantic events such as data races. In this section, we first describe the semantic domains and then proceed to discussing our problem formulation.

\[
O: \text{ Domain of abstract objects (i.e., allocation sites),} \\
V: \text{ Domain of variable identifiers,} \\
F: \text{ Domain of instance field identifiers,} \\
M: \text{ Domain of methods,} \\
C: \text{ Domain of call sites,} \\
\text{Inv} \in M \times C: \text{ Relation of call sites and their target methods,} \\
\text{Con} \in M \times C: \text{ Relation of methods and their contained call sites.}
\]

An event is a quintuple \( e = (o, v, f, stmt, op) \), where \( o \in O \), \( v \in V \) is a variable that points to \( o \), \( f \in F \) is the concerned field of \( o \), \( stmt \) is a heap access statement in the form of \( a = v.f \) or \( v.f = b \), and \( op \in \{r, w\} \) indicates whether it is a read (heap load) or a write (heap store). In other words, \( v \) is the variable accessed in the load/store statement \( stmt \) and \( o \) is the object \( v \) points to. In different dynamic analyses, an event may carry different semantics. For example, for a data race detector, an event is a load/store that participates into a race condition, while a memory leak detector such as Sleigh [8] reports the last use site (a load or a store) of a leaking object as an event. For both detectors, \( o \) is the allocation site of the object accessed, also reported for problem diagnosis. Using \( \text{pointsto}(v) \) to denote the set of objects \( v \) points to, we define the following two types of contexts.

**Definition 3.1** (Calling Context). A calling context is a vector of method-call-site pairs in the form of \((m_1, c_1), (m_2, c_2), \ldots, (m_n, \bot)\), where \( m_n \in M \) is the last method in the vector, \( m_i \in M, c_i \in C, (c_i, m_{i+1}) \in \text{Inv} \), and \((m_i, c_i) \in \text{Con} \) for all \( i \in [1, n] \).

**Definition 3.2** (Reference Context). A reference context is a vector of object-field pairs in the form of \((o_1, f_1), (o_2, f_2), \ldots, (o_n, \bot)\), where \( o_n \) represents the last object in the vector, \( o_i \in O \) and \( f_i \in F \) for all \( i \in [1, n] \).

Each reference relationship \( o_i \xrightarrow{f_i} o_{i+1} \) is induced by a heap store \( v_i.f = v_{i+1} \) or a heap load \( v_{i+1} = v_i.f \), where \( o_i \in \text{pointsto}(v_i) \) and \( o_{i+1} \in \text{pointsto}(v_{i+1}) \).

For simplicity, we will use the chain-based forms to represent a calling context \((m_1 \xrightarrow{c_1} m_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} m_n)\) and a reference context \((o_1 \xrightarrow{f_1} o_2 \xrightarrow{f_2} \ldots \xrightarrow{f_{n-1}} o_n)\).

**Example 3.3** (Calling and Reference Contexts). Consider Figure 2 again. We omit the calling context for method \( \text{runCompare()} \) for brevity. For the last use site of object \( o_2 \) at line 14, we have a calling context \( \xrightarrow{c_1} \text{runCompare()} \xrightarrow{c_2} \text{value()} \). The reference context for \( o_2 \) is \( o_9 \xrightarrow{f_1} o_{array} \xrightarrow{arr_elm} o_{2} \). Both the calling and reference contexts are closely related to the underlying symbolic points-to graph. For instance, the reference context corresponds to a path \( p_r: S_1 \xrightarrow{array} S_2 \xrightarrow{arr_elm} S_3 \) in Figure 2b, where \( S_1 \) and \( S_2 \) are the symbolic objects pointed to by the variable nodes \( \text{this}_{19} \) and \( \text{tmp}_{19} \), respectively. The calling context corresponds to a path \( o_2 \xrightarrow{call_{4}} S_{13} \). The goal of our analysis is to find all such reference paths \( p_r \) using (1) the variable accessed in the event (i.e., \( in_{13} \)) as the starting point and (2) the calling context as a constraint (i.e., the search for \( p_r \) has to go through edge \( o_2 \xrightarrow{call_{4}} S_{13} \)).

3.2 Problem Statement

Given an event \( e = (o, v, f, stmt, op) \) of interest, a dynamic analysis collects a calling context \( c \) such that the last method \( m_n \in c \) contains the statement \( stmt \in e \). In most cases, the calling context is a full stack trace starting from \( \text{main} \). We represent event \( e \) and its corresponding calling context \( c \) as a calling-context-event (CE) pair \((c, e)\). Similarly, we define a reference-context-event (RE) pair \((p, e)\) such that the object \( o \) accessed by statement \( stmt \in e \) is the last object \( o_n \in c \) of the reference path \( p \). We say a CE pair \((c, e)\) is consistent with a RE pair \((p, e)\) iff the sequence of method calls/returns executed to generate the reference path \( p \) matches the call stack \( c \). This matching essentially boils down to checking a string prefix relationship, which will be discussed in Section 4. A consistent RE pair \( p \) abstracts a dynamic heap reference path leading to the object in \( e \) at the moment \( e \) is captured. Our goal is thus to statically find all such RE pairs that are consistent with a given \((c, e)\) captured by the dynamic analysis.
We model the calling and reference contexts using the work [28, 31]. This section discusses the constraint-guided formulation of Context Translation. Given a string s over the alphabet $\Sigma_M \cup \Sigma_H$, the $M$-component of s is a string transformed by removing all brackets, respectively. We use alphabet $\Sigma_M$ to denote method calls and alphabet $\Sigma_H$ to denote heap accesses. Sridharan et al. [37] employ the following treatments to model a context-sensitive object flow:

- Each method call entry is treated as “(” in alphabet $\Sigma_M$, where i is the ID of a call site. Similarly, each method return exit is treated as “)”. In alphabet $\Sigma_M$.

- Each heap load of field $f$ is treated as “[ ” and each heap store to $f$ is treated as “] ” in alphabet $\Sigma_H$.

Given a string $s$ over the alphabet $\Sigma = \Sigma_M \cup \Sigma_H$, the $M$-component of $s$ is a string formed by removing all brackets, respectively. We use alphabet $\Sigma_M$ to denote method calls and alphabet $\Sigma_H$ to denote heap accesses. Sridharan et al. [37] employ the following treatments to model a context-sensitive object flow:

- Each method call entry is treated as “(” in alphabet $\Sigma_M$, where i is the ID of a call site. Similarly, each method return exit is treated as “)”. In alphabet $\Sigma_M$.

- Each heap load of field $f$ is treated as “[ ” and each heap store to $f$ is treated as “] ” in alphabet $\Sigma_H$.

Method calls and heap accesses are modeled using the two CFLs $M$ and $H$ over $\Sigma = \Sigma_H \cup \Sigma_M$, respectively. The grammars for $M$ and $H$ are specified using the following productions. Specifically, language $M$ generates a string of well-balanced parentheses to model "realizable call paths" [31], shown below as a nonterminal BC. Similarly, nonterminal BF generates a string of well-balanced brackets to model "balanced field accesses".

\[
\text{Language } M : \quad \text{BC} \rightarrow \text{BC} | ( \text{BC} ) | [ f ] | \epsilon, \\
\text{Language } H : \quad \text{BF} \rightarrow \text{BF} | [ f ] | ( ) | \epsilon.
\]

Language $M$ guarantees that all method calls and returns are properly matched on an object flow, while language $H$ ensures that a retrieval from an object field would obtain a value only if the value has been written into the same field. Note that the $H$-reachability formulation also introduces the inverse store and load accesses to soundly approximate object aliasing [37, 38]. That is, for heap access edges of the form $o_x \xrightarrow{\text{store}(f)} o_y$ and $o_x \xrightarrow{\text{load}(f)} o_y$, there exist inverse edges $o_y \xrightarrow{\text{load}(f)} o_x$ and $o_y \xrightarrow{\text{store}(f)} o_x$, respectively. Similarly, for call/return edges of the form $o_x \xrightarrow{\text{call}} o_y$ and $o_x \xrightarrow{\text{return}} o_y$, their inverse edges $o_y \xrightarrow{\text{return}} o_x$ and $o_y \xrightarrow{\text{call}} o_x$ exist as well.

In the context of SPG [45], the treatment of open ("["") and close ("]") brackets is slightly different than in [37]. As
discussed earlier, in an SPG, loads of the form \( a = b.f \) and stores of the form \( b.f = a \) are both abstracted by a single points-to edge \( S_b \xrightarrow{f} S_a \). Hence, the question of whether two variables are pointer aliases reduces to understanding whether the symbolic nodes or objects they point to are memory aliases. The MemAlias formulation introduced in [45] is the same as the \( \mathcal{H} \)-reachability except that the close and open brackets represent a generic field points-to edge \( f \) and its inverse edge \( f^{-1} \), respectively. Since the graph is bidirected, if there is an edge \( S_b \xrightarrow{f} S_a \), the inverse edge \( S_a \xleftarrow{f} S_b \) exists automatically.

**Example 4.1 (Flow-Insensitive Alias Analysis via \( \mathcal{H} \)-Reachability).** We consider the following Java-like program: “\( a = b.f; b = c.g; c.g = e; e.f = h \)”. Under the SPG, they generate four points-to edges \( S_b \xrightarrow{f} S_a, S_c \xrightarrow{g} S_b, S_c \xrightarrow{g} S_e \), and \( S_e \xrightarrow{f} S_h \). Treating \( \xrightarrow{f} \) as open brackets, \( \xleftarrow{f} \) as close brackets, \( \xrightarrow{g} \) and \( \xleftarrow{g} \) as open brackets, there exists a MemAlias-path from \( S_b \) to \( S_a \); \( S_b \xrightarrow{f} S_c \xrightarrow{g} S_b \xrightarrow{g} S_e \xrightarrow{f} S_h \rightarrow S_a \), with balanced open and close brackets. This indicates that \( S_b \) and \( S_a \) are memory aliases, and thus the variables \( a \) and \( h \) are pointer aliases.

**Language \( \mathcal{M} \cap \mathcal{H} \) for Object Flow.** In the literature, static analyses [37, 45, 46] typically use the language \( \mathcal{M} \cap \mathcal{H} \) to model the context-sensitive object flow, ensuring the well-balanced property from both \( \mathcal{M} \) and \( \mathcal{H} \). Unfortunately, CFLs are not closed under intersection [51]. In addition, the precise \( \mathcal{M} \cap \mathcal{H} \)-reachability problem is shown to be undecidable [29]. In practice, we often assume the absence of recursive calls [45, 46] (i.e., recursion is handled context-insensitively). The number of open parentheses in a string is bounded, and thus language \( \mathcal{M} \) becomes a regular language. Since CFLs are closed under intersection with a regular language, the resulting language \( \mathcal{M} \cap \mathcal{H} \) is still a CFL. Therefore, the \( \mathcal{M} \cap \mathcal{H} \)-reachability is an instance of CFL-reachability and a sound approximation of object flow.

### 4.2 Exploiting Unbalanced Parentheses/Brackets

For a CR(n) translation, requiring balanced method calls and heap accesses is often too strict. To illustrate, consider the aforementioned path \( l \) (i.e., \( \text{in}_{13} \rightarrow S_{13} \rightarrow o_2 \xrightarrow{\text{calls}} S_4 \rightarrow S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow o_3 \) ) from the variable node \( \text{in}_{13} \) (i.e., the variable contained in the last use site) to the object node \( o_3 \) (i.e., the container object) in Figure 2b. As discussed in Section 2, this path contains the reference context for the object \( o_2 \) and is what we intend to find. The \( M \)- and \( H \)-components of the label sequence on this path are, respectively, “\( \text{in}_{13} (o_1) o_2 \)” and “[arr(elm array)” where both the parentheses and the brackets are **unbalanced**. Intuitively, the unbalanced parentheses (i.e., method calls/returns) on the path reveal (part of) the call stack while the unbalanced brackets reveal the reference path. In this section, we exploit the unbalancedness in parentheses/brackets to perform context translation.

**Modeling Unbalanced Method Calls/Returns.** Here we first discuss how to model the unbalancedness in method calls/returns on a regular object flow. Consider an event \( e = (o, f, v, stmt, op) \) that \( \text{CR}(n) \) takes as input. The object and variable of interest in the event are \( o \) and \( v \), respectively, and \( v \) points to \( o \). Suppose \( o \) and \( v \) are in the methods \( M_o \) and \( M_v \), respectively. In terms of their relationship on the call graph, there are three possible scenarios: (1) \( M_o \) directly or transitively calls \( M_v \); (2) \( M_o \) directly or transitively calls \( M_v \); and (3) \( M_o \) and \( M_v \) are called directly or transitively by another method \( M_t \). Table 1 depicts these three scenarios, their examples, and the grammar rules that are used to capture them. We discuss these rules as follows.

(a) **Downward context (DC).** In this case, \( M_o \) invokes \( M_v \). A DC-path indicates that an object \( o \) flows down to a variable \( v \) in a callee method. The string on the path representing the flow contains several **unmatched open parentheses**. Therefore, we construct a new nonterminal DC by introducing open parentheses \( ( \) into nonterminal BC, which represents balanced parentheses.

(b) **Upward context (UC).** In this case, \( M_o \) invokes \( M_v \). A UC-path indicates that an object \( o \) flows up to a variable \( v \) in its caller method. The path string contains several **unmatched close parentheses**. Similarly, we construct a new nonterminal UC by adding additional close parentheses \( ) \) into BC.

(c) **Up-downward context (UDC).** In this case, \( M_o \) and \( M_v \) are called by another method \( M_t \). A UDC-path indicates that an object \( o \) first flows up to its caller \( M_t \) and then flows down from \( M_t \) to a callee of \( M_t \). We construct a new nonterminal UDC by combining UC and DC.

Putting them all together, the unbalancedness in parentheses required by \( \text{CR}(n) \) can be described using a CFL \( \mathcal{M} \), shown in Figure 3a, over the alphabet \( \Sigma = \Sigma_M \cup \Sigma_H \). The main rule starts with a nonterminal UBC, which is a union of DC, UC, and UDC. Note that we do not need a down-upward context (DUC) between \( o \) and \( v \). If \( o \) in method \( M_o \) first flows down to a callee, it has to come back up through the same path to method \( M_v \) or \( M_u \). Therefore, the path would simply reduce to a UC or DC path.

**Figure 3.** Languages \( \mathcal{M} \) and \( \mathcal{H} \) for modeling unbalanced method calls and heap accesses; the nonterminals BC and BF generate the well-balanced language \( \mathcal{M} \) and \( \mathcal{H} \), respectively.
While practical points-to [37] and alias [46] analyses often allow this unbalancedness [18] in their implementations for context sensitivity, our work is the first attempt to formally explore the properties of $M$. Furthermore, none of the existing techniques have explored the possibility of extending $H$ to handle unbalanced brackets, primarily because, in the context of points-to and alias modeling, heap accesses are required to be balanced. In a CR(n) translation, on the contrary, allowing unbalanced brackets in $H$ exposes heap access paths, providing insights for context translation.

**Modeling Unbalanced Heap Accesses.** Consider Example 4.1 again. Suppose that object $o_h$ (the actual object $h$ points to) is the leaking object reported by a dynamic analysis in a CE pair and there is no calling context reported. A CR(2) translation produces a reference context $o_c \rightarrow o_e \rightarrow f \rightarrow o_h$ for $o_h$. This reference path is actually generated by the two heap stores $c.g = e$ and $e.f = h$. In any string accepted by language $H$, these stores are open brackets that must have matched closed brackets (loads) because $H$ is designed to model points-to/aliasing relations that require all brackets to be balanced. For CR(n), however, we need a different language that can expose these unmatched brackets, which are the building blocks of the reference paths to be uncovered.

For this purpose, we extend language $H$ to $\overline{H}$ over the same alphabet $\Sigma = \Sigma_M \cup \Sigma_H$. Figure 3b gives the production rules of language $\overline{H}$. Note that $\overline{H}$ allows only unmatched open brackets rather than unmatched close brackets, because finding objects that reference an event object $o$ requires "climbing up" the reference ladder. This corresponds to traversing inverse field points-to edges under the SPG representation. For example, the $H$-component of the aforementioned path $I$ is $\text{[arr_end]} \text{[array]}$, which contains only open brackets. To expose unmatched open brackets, we introduce a nonterminal UBF that represents the "unbalanced field accesses". Note that UBF $\text{UBF}$ in Figure 3b is essentially the same as nonterminal UC $\text{UC}$ in Figure 3a if we apply a proper bijection between parentheses and brackets.

**Reduced Form of Language** $\overline{M} \cap \overline{H}$. A CR(n) translation leverages language $\overline{M} \cap \overline{H}$ to model unbalanced method calls and heap accesses. However, this language is a *static* modeling of calling and reference contexts. For example, a string $s \in \overline{M}$ represents a *trace* of method executions before reaching the event, with balanced parentheses representing invocations that have finished and unmatched parentheses representing method invocations *still on the stack*. The calling context (*i.e.*, stack trace) obtained by a *dynamic* analysis, by contrast, only contains on-stack methods. To illustrate, consider the calling context of the heap load at line 13 in Figure 2a. Before `useValue()` is invoked at line 4, method `openCompareEditorOnPage()` at line 3 has been called and returned. The call stack captured by the dynamic analysis is `$k_4$", while the call trace ($e \overline{M}$) on the path from $in_{t_3}$ to $o_2$ is "$k_4 (s)_5". In this paper, we consider each captured calling context as a sequence of *close parentheses* starting at the event-containing method. Since $G$ is a bidirected graph, treating it as a sequence of open parentheses would also work (*i.e.*, but needs a symmetric handling).

To bridge the gap between the static and dynamic representations of a call trace, we introduce the *reduced form* of a string in $\overline{M} \cap \overline{H}$. Formally, let $s$ be a string over the alphabet $\Sigma = \Sigma_M \cup \Sigma_H$. The reduced string $R_s$ of $s$ is obtained by recursively removing matched parentheses in its $M$-component as well as matched brackets in its $H$-component. For instance, we have $R_{h_4(s)_3} = R_{h_4(s)_3} = R_{h_4(s)_3} = "s"$. It is straightforward to see the reduced string $R_s$ of any $s$ in language $\overline{M} \cap \overline{H}$ is "$\epsilon\".

### 4.3 Constraint-Guided CFL

This section presents our constraint-guided CFL-reachability formulation of CR(n) translation. Our formulation is based on $\overline{M} \cap \overline{H}$-reachability. Since CFLs are not closed under intersection [51], researchers often assume the absence of recursive calls [45, 46] in practice (*i.e.*, recursion is handled context-insensitively). We use the same handling here — as the number of open parentheses in a string is bounded, language $\overline{M} \cap \overline{H}$ becomes a regular language and thus the language $\overline{M} \cap \overline{H}$ is indeed a CFL. Here we first focus on the formulation of our CR translation and then briefly discuss how constraint-guided CFL-reachability can solve other analysis problems.

For CR(n), our constraint-guided $\overline{M} \cap \overline{H}$-reachability introduces two types of constraints to the traditional $\overline{M} \cap \overline{H}$-reachability problem: a *string constraint* and a *path constraint*. 

<table>
<thead>
<tr>
<th></th>
<th>Downward context (DC)</th>
<th>Upward context (UC)</th>
<th>Up-downward context (UDC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call relation</td>
<td>$M_a \rightarrow \text{call} \rightarrow \ldots \rightarrow \text{call} \rightarrow M_c$</td>
<td>$M_a \rightarrow \text{call} \rightarrow \ldots \rightarrow \text{call} \rightarrow M_c$</td>
<td>$M_a \rightarrow \text{call} \rightarrow \ldots \rightarrow \text{call} \rightarrow M_c$</td>
</tr>
<tr>
<td>Path illustration</td>
<td>$0 \rightarrow \text{l} \rightarrow \ldots \rightarrow \text{l} \rightarrow 1$</td>
<td>$0 \rightarrow \text{l} \rightarrow \ldots \rightarrow \text{l} \rightarrow 1$</td>
<td>$0 \rightarrow \text{l} \rightarrow \ldots \rightarrow \text{l} \rightarrow 1$</td>
</tr>
<tr>
<td>Grammar rules</td>
<td>$\text{DC} \rightarrow \text{BC} \</td>
<td>\ \text{DC} \</td>
<td>\ {f \</td>
</tr>
</tbody>
</table>
Definition 4.2 (String Constraint). Let $t$ be a string over the alphabet $\Sigma_t = \{1, \ldots, k\}$, i.e., $t$ contains only close parentheses, representing a call stack going from the event-containing method to main. Given $t$ and an integer $n$, we consider the following two string constraints defined on a string $s$ over the alphabet $\Sigma = \Sigma_M \cup \Sigma_H$:

- **Prefix constraint ($\pi_t$):** $s$ satisfies the constraint, denoted as $\pi_t \vdash s$, if $s \in \overline{M}$ and the unmatched close parentheses in its reduced string $R_s$ is a prefix of $t$.

- **Unbalanced constraint ($\psi_n$):** $s$ satisfies the constraint, denoted as $\psi_n \vdash s$, if $s \in \overline{H}$ and the number of unbalanced open brackets is $n$.

Definition 4.3 (Path Constraint). Given a digraph $G$ and a particular node $u \in G$, a path $l \in G$ satisfies the path constraint $\alpha_u$ iff $l$ passes $u$, denoted as $\alpha_u \vdash l$.

Intuitively, string $t$ represents the calling context $c$ in a collected CE pair. In a CR($n$) translation, we employ the prefix constraint $\pi_t$ to guarantee that the inferred reference context is consistent with the collected calling context represented by string $t$. The unbalanced constraint $\psi_n$ imposes a length requirement on the inferred reference contexts. Finally, the path constraint $\alpha_u$ ensures that each inferred reference path in the graph actually goes through the object node $u$ accessed in the collected event. We refer to $u$ as the anchor node in the path. Next, we discuss a concrete example.

Example 4.4 (String Constraint). Consider the path $l$ from node $in_{13}$ to $o_9$ in Figure 2. The path goes through node $u = o_2$. Therefore, we have $\alpha_u \vdash l$. The corresponding path string $s_l$ is $"(\{a\} el\{m\} ar\{ray\})00\"$ and its reduced string $R_{s_l}$ is $"(\{\})\"$ (i.e., captured call stack) and $n = 2$ (i.e., length requirement). We can see that the unmatched close parentheses in $R_{s_l}$ is $"(\})\"$ which is a prefix of the calling context $t = "(\})\"$. Moreover, $R_{s_l}$ contains 2 unmatched open brackets, satisfying the unbalanced constraint. As a result, we have $\pi_t \vdash s_l \land \psi_n \vdash s_l$.

In general, constraint-guided CFL-reachability applies string constraints to the path string $s_l$ of each $\overline{M} \cap \overline{H}$ path $l$, while the path constraint is applied to the path $l$ itself. Next, we formally define our formulation.

Definition 4.5 (Constraint-Guided CFL-Reachability). Given an edge-labeled digraph $G$, a constraint-guided-CFL (CG-CFL) reachability problem is denoted as $\overline{L}_\phi^\alpha$-reachability, where $\overline{L}$ is a CFL, $\phi$ is a string constraint, and $\alpha$ is a path constraint. Node $u \in G$ is $\overline{L}_\phi^\alpha$-reachable from $u$ iff there exists a path $l$ from $u$ to $v$, such that $l$ satisfies the path constraint $\alpha$ (i.e., $\alpha \vdash l$) and its path string $s_l$ of $l$ satisfies the string constraint $\phi$ (i.e., $\phi \vdash s_l$).

Definition 4.6 (CG-CFL Formulation of CR($n$)). Given a digraph $G$, a dynamically captured CE pair ($c$, $e = (o, v, f, stmt, op)$), as well as a user-defined length parameter $n$, a CR($n$) translation is formulated as a single-source $\overline{M}_\alpha^\phi \cap \overline{H}_\alpha^\psi$-reachability problem in $G$, which aims to find all object nodes $o' \in G$ such that node $o'$ is $\overline{M}_\alpha^\phi \cap \overline{H}_\alpha^\psi$-reachable from the variable node $v$ (contained in the event statement). Specifically, each such path $l$ from $v$ to $o'$ satisfies the path constraint $\alpha_u \vdash l$. The path string $s_l$ satisfies the string constraint $\pi_c \vdash s_l \land \psi_n \vdash s_l$.

Informally, a CR($n$) translation uses the recorded event object $o$ as the anchor node and imposes a prefix constraint $\pi_c$ w.r.t. the recorded context $c$ in the exploration of the $\overline{M}$-paths while looking for a sequence of up to $n$ unbalanced open brackets (i.e., satisfying the unbalanced constraint $\psi_n$) in the exploration of $\overline{H}$-paths. Using $o$ as the anchor node, an $\overline{M}_\alpha^\phi \cap \overline{H}_\alpha^\psi$-path $l = v \rightarrow \ldots \rightarrow o'$ can be naturally divided into two sub-paths w.r.t. the midpoint $o$. We denote them as $l_1 = v \rightarrow \ldots \rightarrow o$ and $l_2 = o \rightarrow \ldots \rightarrow o'$, respectively. Consider their corresponding path strings $s_{l_1}$ and $s_{l_2}$.

- **String $s_{l_1} \in \overline{M} \cap \overline{H}$ models the inverse of the object flow from $o$ to $v$. Since $o$ and $v$ are both captured by the dynamic analysis and $v$ points to $o$, $s_{l_1}$ does not contain any unbalanced heap accesses.**

- **String $s_{l_2} \in \overline{M} \cap \overline{H}$ contains open brackets explaining how $o$ is referenced (directly or transitively) by $o'$.**

Since object $o$ can be referenced by many other objects, only a few of which are relevant for the captured event, the essence of this formulation is to use prefix constraints to filter out irrelevant paths that are not consistent with the calling context $c$ captured. In addition, since the calling context is w.r.t. the event statement (containing the variable $v$), we use $v$ as the starting point for the path search.

CR($n$) decomposes $l$ into $l_1$ and $l_2$ to compute the $\overline{M}_\alpha^\phi \cap \overline{H}_\alpha^\psi$-reachability. In particular, the first sub-path $l_1$, without unbalanced heap accesses, could be computed using a context-sensitive alias analysis such as [46]. We can find $l_2$ by modifying the alias analysis to identify unbalanced open brackets that are consistent with $l_1$.

Example 4.7 (CR($n$) Translation via CG-CFL-Reachability). Figure 4a shows a slightly more complex example that we use to illustrate our translation. This example has many interesting properties of a typical object-oriented program, such as use of collections and factory methods (e.g., createList()) that Figure 2a does not have. During its execution, an event $e$ occurs when $a$.value is read on line 21. Suppose a dynamic analysis records a CE pair $(c, e)$ where $c = foo(o)\rightarrow main(o)$ and $e = (o_{15}, a, value, stmt_{21}, r)$. Here $call_b$ indicates the call site at line 6 invoking $foo()$, $stmt_{21}$ represents the read statement that occurs at line 21, and $o_{15}$ represents the allocation site at line 15.

Figure 4a shows that variable $m_2$ at line 2 holds a reference to object $o_{15}$ by calling method populateList(). Figure 4b shows the corresponding graph representation. From the graph, we can see that variable $m_2$ is an alias of $r_{10}$, which
5 Translation Algorithm

This section discusses our CR(n) translation algorithm based on the SPG representation [45] of the program.

5.1 Symbolic Points-to Graph and MemAlias

An SPG is a locally-resolved points-to graph: it contains real points-to relationships that can be resolved within a method while using placeholder symbolic nodes to represent objects that are created outside of the method. SPG simplifies the complex object flow between an object and a variable (through copy assignments, parameter passing, etc.) into simple MemAlias relationships between the object and a symbolic node, a feature that simplifies our analysis. An SPG is constructed entirely by an Andersen-style analysis for each method. Different SPGs are connected (trivially) at call sites to form an ISPG for the program. We refer the reader to [45] for the details of its construction.

As briefly discussed in Section 4, MemAlias \( \subseteq (O \cup S) \times (O \cup S) \) [45] is defined in the same way as language \( \mathcal{H} \) (cf. Figure 3b) with the close and open brackets representing a field points-to edge \( f \) and its inverse edge \( \overline{f} \). The context-sensitive MemAlias relation used in our CR(n) translation is expressed using \( \mathcal{M} \cap \mathcal{H} \)-reachability over an ISPG. As with other context-sensitive analyses, it allows unbalanced method calls for context-sensitivity but does not need unbalanced heap accesses due to its modeling of aliases.

SPG allows for the answering of alias queries without first performing a points-to-analysis. For example, if we have two statements \( a = b.f \) and \( d = b.f \), there will be two points-to edges \( S_9 \rightarrow S_{10} \) and \( S_8 \rightarrow S_9 \). The MemAlias grammar can quickly find that the symbolic nodes \( S_a \) and \( S_d \) are memory aliases without knowing which objects \( b \) may point to. This formulation is particularly suitable for us for two reasons: (1) the value flows via local copy assignments and parameter passing have been eliminated; field accesses are modeled explicitly using \( f \) and \( \overline{f} \) edges, making it easier for us to match references and (2) since stores and loads are unified by field edges \( f \), strings in MemAlias become palindromes (e.g., \( abcba \)), leading to improved efficiency.

Example 5.1 (The \( \mathcal{M} \cap \mathcal{H} \)-Path). In Figure 4b, there exists an \( \mathcal{M} \cap \mathcal{H} \)-path from \( S_6 \) (i.e., the placeholder of the event object) to \( o_{15} \) (i.e., the actual event object created in \( \text{populateList} \)): \n
\[
\begin{align*}
S_6 \xrightarrow{f} S_7 \xrightarrow{f} S_8 \xrightarrow{\overline{f}} S_9 \xrightarrow{f} S_{10} \xrightarrow{f} S_{11} \xrightarrow{f} S_{12} \xrightarrow{f} S_{13} \xrightarrow{f} S_{14} \xrightarrow{f} S_{15} \xrightarrow{f} S_{16} \xrightarrow{f} S_{17} \xrightarrow{f} o_{15}.
\end{align*}
\]

\(\) is the symbolic node that is a memory-alias of the event object \( o_{15} \).

Figure 4. A more complicated example and its ISPG.

(a) Example code.

```java
1 void main(String[] args){
2     List m = createList();
3     populateList(m);
4     for(int i = 0; i < m.size(); i++){
5         A e = (A) m.get(i);
6         foo(e);
7     }
8 }
9 List createList(){
10     List r = new ArrayList();
11     return r;
12 }
13 void populateList(List m){
14     for(int i = 0; i < m.size(); i++){
15         A a = new A();
16         m.add(a);
17     }
18     void foo(A a){
19         //read event
20     }
21     println(a.value);
22 }
```

(b) ISPG for the example: nodes are named in the same way as in Figure 2b; \( S_6 \) is the symbolic node that is a memory-alias of the event object \( o_{15} \).
Algorithm 1: Algorithm for CR(n) translation.

Input: An ISPG G, a user parameter n, a CE pair (c, e = (o, v, . . . ))
Output: A set R of reference contexts [o_m u = a_n−1 f_n−1 . . . f_1 = o]
1 initialize set R to Ø
2 initialize worklist W to Ø
3 let set SN be the set of symbolic nodes pointed to by c in G
4 foreach S_o ∈ SN do
5 construct a string str from the calling context c
6 T_M ← MemAlias(S_o, o, str)
7 foreach t_M ∈ T_M do
8 initialize an empty stack t_H
9 insert to W the tuple (o, t_M, t_H, false)
10 while W ≠ Ø do
11 remove a tuple (x, t_M, t_H, b) from W
12 if x ∈ S & b then
13 pairs ← MemAliasAll(x, str)
14 foreach (o', t_H) ∈ pairs do
15 foreach t_M ∈ T_M do
16 t_H ← t_H
17 let (o', f) be the top of stack t_H
18 pop t_H and push (o', f) onto t_H
19 if o' is unvisited under the context t_M then
20 W ← W ∪ ((o', t_M', t_H, false))
21 else
22 /*Inspect x's edges*/
23 if |t_M| > 0 then /*A solution is found
24 Return t_M
25 if |t_M| = n then /*don't search any more
26 continue
27 foreach edge u ← x ∈ G do
28 t_M' ← t_M, t_H ← t_H and b ← b
29 if label = f then
30 push (u, f) onto t_H
31 V ← true
32 else if label = c then
33 if top of t_M' is t_M, then
34 pop t_M
35 else //unrealizable call chain
36 continue
37 else if label = l then
38 push t_M onto t_M
39 if u is unvisited under the context t_M then W ← W ∪
40 (u, t_M', t_H, b)
41 return R

The H-component of the path string $”arr_elm$ array array$” belongs to language $H$, indicating that the variables q_{15} and q_{19} pointing to these two nodes respectively are pointer aliases.

5.2 CR(n) Translation

This subsection describes the CR(n) translation algorithm that computes the single-source-multiple-sink $M$ ∩ $H$-reachability based on MemAlias.

**Basic Idea.** Algorithm 1 shows the CR(n) translation algorithm that solves single-source $M$ ∩ $H$-reachability. As described in Section 4.3, in a CR(n) translation, each $M$ ∩ $H$-path from the variable node $v$ to an object node $o'$ is composed of two parts, i.e., the first part $l_1$ from $v$ to the event object $o$ and the second part $l_2$ from $o$ to $o'$. The basic idea of the algorithm is to handle these two sub-paths in two phases. In the first phase (lines 5 – 6), we compute $l_1$ from $v$ to $o$. Since $v$ points to $o$, the symbolic node $S_o$ (that $v$ points to) must be a memory alias of $o$. Hence, $l_1$ has essentially been captured by the MemAlias formulation (between $S_o$ and $o$).

In the second phase (lines 10 – 39), we explore the graph starting from node $o$. Due to the bidirectedness of the ISPG, we only need to traverse the incoming edges of each node to identify path $l_2$. During the graph traversal, for each incoming edge $u \rightarrow x$ of node $x$, we employ two stacks $t_M$ and $t_H$ to handle method-call and field-access edges, respectively.

- **Handling of call/return edges:** for each return edge $u \rightarrow x$, we push a symbol "(i)" onto stack $t_H$. We pop symbol "(i)" when encountering a matched call edge $u \rightarrow x$.

- **Handling of field points-to edges:** for each edge $u \rightarrow x$, we push the node-field pair $(u, f)$ onto stack $t_H$.

The stack operations for handling call/return edges are defined in expected ways. We use the alias analysis algorithm MemAlias, proposed in [45], to test whether two (object and symbolic) nodes are memory aliases. The original MemAlias algorithm works as follows: it takes as input two symbolic or object nodes $S_o$ and $S_v$, and returns all $M$ ∩ $H$-paths from $S_o$ to $S_v$. If $S_o$ and $S_v$ are not aliases, it returns an empty set.

In this work, we extend the MemAlias algorithm in a way so that it takes an additional input string $str$, which corresponds to the $M$-component of the reduced form of a string in language $M$ ∩ $H$. Initially, $str$ represents the calling context $c$ collected in a CE pair (line 6). The extended MemAlias algorithm achieves two goals. First, it ensures that any MemAlias-path $l$ it finds from $S_o$ to $o$ has to be consistent with $str$. Second, it returns a set of stacks $T_M$, each stack $t_M$ of which represents the $M$-component of the reduced form of $l$'s path string.

Based on MemAlias, we derive a new algorithm called MemAliasAll (line 14), which solves single-source $M$ ∩ $H$-reachability without needing the target node. Given a source symbolic node $S_o$, and a string $str$ representing the $M$-component of the reduced form of a string in language $M$ ∩ $H$, MemAliasAll $(S_o, str)$ finds all such (symbolic or object node) $o$ that there exists an $M$ ∩ $H$-path from $S_o$ to $o$ and the path string is consistent with $str$. MemAliasAll returns a set of node-stack pairs in the form of $(o, t_M)$, each of which contains an object $o$ that is a memory alias of $S_o$ and a stack $t_M$, representing the $M$-component of the reduced form of a path from $S_o$ to $o$.

The basic structure of the algorithm is as follows: (1) we query MemAlias (line 6) for the set of $M$ ∩ $H$-paths from $S_o$ (the symbolic node variable $v$ points to) to the event object $o$, using the collected calling context $c$ as a string constraint. Each path found by MemAlias thus represents the first sub-path $l_1$, which is consistent with $c$. Each stack $t_M$ returned by MemAlias represents the $M$-component of the reduced form of $l_1$, which will be used subsequently to find the second sub-path; (2) we search the graph starting from $o$ for the second
sub-path $l_2$, using $t_M$ as a starting context; hence, each $l_2$ found needs to be consistent with $t_M$. In this process, we keep searching for field points-to edges of the form $a \rightarrow b$ that are building blocks of a reference path. One tricky case here is that since ISPG is a partially resolved points-to graph, many symbolic nodes may represent the same object. Hence, when we reach a symbolic node $S_1$ (backward) from an edge $S_1 \rightarrow S_2$, it is not enough to only follow the incoming edges of $S_1$ itself in the next step. Since $S_1$ may be memory aliases of other nodes such as $S_3$ and $o_4$, it is important to find these nodes and follow their incoming edges as well. To this aim, we use algorithm MemAliasAll (line 14).

**Main Algorithm.** This algorithm takes as input a CE pair and a user parameter $n$ and returns a set of reference contexts $R$. For each symbolic node $S_v$ that variable $v$ points to, we first query algorithm MemAlias for finding all $M^x_f \cap H$ paths from $S_v$ to the event object $o$ (line 6). MemAlias returns a set of stacks $T_M$ for the paths found. These paths are essentially the $l_1$ sub-paths as discussed above. The goal of the rest of the algorithm is to find all sub-paths $l_2$ such that the concatenated path string $"s_l \oplus s'_l"$ satisfies the constraints $\pi_c$ and $\psi_n$.

The algorithm maintains a worklist $W$ and iteratively adds tuples into $W$. Each iteration of the loop on line 11 retrieves a tuple in the form of $(x, t_M, t_H, b)$, where $x$ is the node being processed, $t_M$ and $t_H$ are the two stacks for graph traversal, and $b$ is a boolean flag indicating whether node $x$ is reached by following a field edge. If flag $b$ is set, we query the MemAliasAll algorithm to find the alias set of the current node. Each pair $(o_i, f_i)$ in $t_H$ indicates that object $x$ being processed is reachable from field $f_i$ of the object $o_i$. Our algorithm explores the ISPG to gradually find each $(o_i, f_i)$ leading to the event object $o$.

Initially, the event object $o$ is added to the worklist $W$ with the stack $t_M$, which is obtained from MemAlias at line 6. $t_M$ is used as the initial constraint to guide the graph traversal (line 7–9). Each iteration removes a tuple $(x, t_M, t_H, b)$ from $W$ (line 11). Suppose the top of stack $t_H$ is $(o_i, f_i)$, the goal of the iteration is to find another object $o_{i+1}$ such that the reference relationship $o_{i+1} \rightarrow o_i$ holds. Node $x$ being processed may be either a symbolic node or an object node.

**Lines 12–20:** If $x$ is a symbolic node and $b$ is true indicating a new field edge has been recently added to $t_H$, we query MemAliasAll to find all objects $o'$ that are memory aliases of $x$. MemAliasAll also returns, for each $o'$, a set $T'_M$ of stacks under which the aliasing occurs. For each stack $t'_M \in T'_M$, we add a tuple $(o', t'_M, t_H, false)$ to the worklist for further processing.

**Lines 21–39:** If $x$ is an object node, we need to explore the field edges to find objects that can reference $x$. The first step is to check whether the size of the stack $t_H$ already reaches the user-defined length parameter $n$ (line 25). If it does, there is no need to search any more and we add the stack $t_H$ to the solution set $R$. Otherwise, we traverse $x$’s incoming edges (lines 27–39). We are particularly interested in incoming edges of the form $u \rightarrow x$: if such an edge is encountered, we push a new pair $(u, f)$ onto $t'_H$. We then set $b'$ to true, indicating that a new field edge has been found. This flag triggers the invocation of MemAliasAll (lines 12–20) in future iterations. All the other branches deal with method entry/exit edges to guarantee calling context sensitivity. Finally, a new tuple is added to the worklist (line 39).

Next, we give an example to illustrate the major steps involved in our CR($n$) translation algorithm.

**Example 5.2 (CR($n$) Translation on the ISPG).** Given the CE pair $(c = \text{foo} \rightarrow \text{main}, e = \langle o_{15}, o_{19}, \ldots \rangle)$ for the running example in Figure 4a, we show how Algorithm 1 translates reference contexts using the corresponding ISPG in Figure 4b. First, since variable $a_{19}$ points to $S_8$ on the graph and $o_{15}$ is the object captured by the dynamic analysis, we know that there must exist a MemAlias-path from $S_8$ to $o_{15}$. So we query MemAlias (line 6) to find the first sub-path $l_1$ from $S_8$ to $o_{15}$ using $c$ as the constraint. Example 5.1 has already given such a path. Its path string $s_1$ clearly satisfies the constraint $\pi_c$, i.e., $\pi_{\text{true}} \vdash s_1$. Because there is only one such path, the stack set $T_M$ returned by MemAlias contains only one stack $t_M = \langle "o\text{'}", "e\text{'}", "f\text{'}" \rangle$, which corresponds to the reduced M-component $h_X(\text{"e"})$ of the path string $s_1$. A tuple $(o_{15}, t_M, \emptyset, false)$ is thus added to worklist $W$ (line 9) and is processed in the first iteration of the loop on line 10.

Since $o_{15}$ is an object node, we traverse its incoming edges to find the second sub-path $l_2$. Node $o_{15}$ has one incoming edge $S_{11} \rightarrow o_{15}$. Hence, we add a new tuple $(S_{11}, t'_M, \emptyset, false)$ to the worklist, where $t'_M = \langle "o\text{'}", "e\text{'}", "f\text{'}" \rangle$. When this tuple is processed in the next iteration, we skip the MemAliasAll query on lines 12–20 since $b$ is false and proceed to inspecting $S_{11}$’s incoming edges. Node $S_{11}$ has one incoming field points-to edge $S_{12} \rightarrow \text{arr(elm)} \rightarrow S_{11}$ and $S_{12}$ has not been visited yet. We process this edge (lines 29–31) and add the tuple $(S_{12}, t'_M, t'_H, true)$ to the worklist (line 39) where $t'_M = \langle "o\text{'}", "e\text{'}", "f\text{'}" \rangle$ and $t'_H$ contains only $(S_{12}, \text{arr(elm)})$.

The next iteration of the loop retrieves this tuple and queries MemAliasAll on $S_{12}$ using $t'_M$ as the string constraint $str$ at line 14. MemAliasAll returns two node-stack pairs w.r.t. $S_8$ and $o_{27}$. Consider the returned pair $(o_{27}, T''_M)$, where $T''_M$ contains only one stack $t''_M = \langle "o\text{'}", "e\text{'}", "f\text{'}" \rangle$. Lines 17–20 replace the top of stack $t'_H$ with $(o_{27}, \text{arr(elm)})$, which generates a new stack $t''_H$, and then add the tuple $(o_{27}, t'_M, t''_H, false)$ into worklist $W$. Finally, the check on line 23 returns true and reports a reference context $o_{27} \rightarrow \text{arr(elm)} \rightarrow o_{15}$ of length 1. Similarly, further processing of the worklist tuple $(o_{27}, t'_M, t''_H, false)$ obtains another reference context $o_{10} \rightarrow o_{27}$. As a result, the CR(2) translation has found
which could not execute a few programs in the most recent version of DaCapo (9.12).

To assess the cost and precision of our analysis, we took the 20 most stale objects for each program reported by the Sleigh leak detector in CCU. These 20 objects, their allocation sites, and the calling contexts of their last use sites were analyzed. We used 20K in our experiments — when the budget is encountered, our analysis stops the attempt to "grow" any paths and reports the set of paths found.

Note that we have also run our static analysis on the Pacer reports. These results are not shown due to space limitations. In addition, while Pacer and Sleigh are two different dynamic analyses, from the perspective of static analysis, there is no difference between them — our static analysis analyzes the same (DaCapo) programs; the heaps for each program execution from Sleigh and Pacer are also the same. The numbers for Pacer are very similar.

Table 2 reports two major measurements: (1) analysis performance — the average time $T$ taken to answer each query (in seconds), the length of dynamic calling contexts $CL$ averaged across the queries, the average number of methods $TM$ traversed to answer each query, the ratio $NR$ between the numbers of the static and dynamic reference paths, and the ratio $LR$ between the maximum lengths of the dynamic and static reference paths.

<table>
<thead>
<tr>
<th>Bench</th>
<th>M(K)</th>
<th>T(s)</th>
<th>CL</th>
<th>TM</th>
<th>NR</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>12.9</td>
<td>11.6</td>
<td>19</td>
<td>438</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>bloat</td>
<td>10.8</td>
<td>16.4</td>
<td>16</td>
<td>188</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>chart</td>
<td>17.4</td>
<td>27.6</td>
<td>16</td>
<td>80</td>
<td>1.0</td>
<td>2.8</td>
</tr>
<tr>
<td>hsqldb</td>
<td>12.5</td>
<td>13.9</td>
<td>23</td>
<td>1628</td>
<td>1.3</td>
<td>1.7</td>
</tr>
<tr>
<td>luindex</td>
<td>10.7</td>
<td>5.2</td>
<td>8</td>
<td>16</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>lusearch</td>
<td>10.2</td>
<td>8.9</td>
<td>15</td>
<td>1413</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>fop</td>
<td>23.5</td>
<td>16.7</td>
<td>8</td>
<td>23</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>pmd</td>
<td>15.3</td>
<td>29.1</td>
<td>50</td>
<td>796</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>jython</td>
<td>27.5</td>
<td>12.4</td>
<td>2</td>
<td>14</td>
<td>1.0</td>
<td>4.5</td>
</tr>
<tr>
<td>xalan</td>
<td>12.8</td>
<td>14.8</td>
<td>5</td>
<td>14</td>
<td>1.0</td>
<td>2.7</td>
</tr>
<tr>
<td>eclipse</td>
<td>41.0</td>
<td>10.4</td>
<td>10</td>
<td>187</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td><strong>geomean</strong></td>
<td><strong>13.7</strong></td>
<td><strong>1.02</strong></td>
<td><strong>1.56</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Analysis performance and precision on DaCapo-2006: we report, for each program, the total number $M$ of statically reachable methods, the analysis time $T$ averaged across the translation queries (in seconds), the length of dynamic calling contexts $CL$ averaged across the queries, the average number of methods $TM$ traversed to answer each query, the ratio $NR$ between the numbers of the static and dynamic reference paths, and the ratio $LR$ between the maximum lengths of the dynamic and static reference paths.

Our implementation includes a Soot-based analysis implementation and the modification of CCU [14] on JikesRVM that enables heap dumping for two real dynamic analyses implemented in CCU — a leak detector Sleigh [8] and a race detector Pacer [7]. To demonstrate the effectiveness of our translation, we conducted two sets of experiments. The first experiment focused on understanding how our analysis performs under different parameters $n$ based on the DaCapo benchmark set [6].

The second experiment, which contained studies focused on Sleigh and Pacer, was designed to assess whether the translated contexts could help developers better understand the reported problems. All experiments were executed on a machine with an Intel Xeon E5620 2.40GHz processor, running Linux 2.6.18. The maximal heap size specified for each program run was 2GB. We used the 2006 version of DaCapo because CCU was built on an old version of JikesRVM (3.1.1), which could not execute a few programs in the most recent version of DaCapo (9.12).

### 6.1 Static Analysis

To assess the cost and precision of our analysis, we took the 20 most stale objects for each program reported by the Sleigh leak detector in CCU. These 20 objects, their allocation sites, and the calling contexts of their last use sites were reported together. Since we modified CCU to dump the heap (discussed in Section 1), we were able to find the complete set of reference paths leading to any leaking object offline in the heap dump. These 20 objects were fed to our static analysis for context translation. The call chain in each sample was used as analysis input and the dynamic reference paths (which are complete) in the heap dumps were used as an oracle, against which the results of our analysis are compared to understand the analysis precision. We set $n$ to a very large number (i.e., 10) to let our analysis find the longest possible paths under a budget (i.e., the maximum number of edges traversed). We used 20K in our experiments — when the budget is encountered, our analysis stops the attempt to "grow" any paths and reports the set of paths found.

We make several observations on these numbers. First, the translation time is reasonably small — due to the demand-driven nature of the analysis, the amounts of time needed to answer queries are all within 30 seconds. Clearly, our technique significantly reduces the runtime overhead of a dynamic analysis since a heap dump typically takes dozens of minutes to several hours.

Second, the quality of the statically reported paths is high — this is reflected by the closeness ($1$) between the numbers of static and dynamic reference paths (i.e., $NR$) and ($2$) between the maximum lengths of the static and dynamic paths (i.e., $LR$). Specifically, the geomeans of $NR$ and $LR$ are $1.02$ and $1.56$, respectively, indicating that the statically inferred reference paths are close to their dynamic counterparts.

The $LR$ of xalan is an outlier. The reason why our static analysis could not find longer reference paths was due to the massive use of recursive data structures to represent XML attributes. The heap writes that initialize these attributes are all in loops. Similarly to [45], we soundly model recursive data structures with artificially created wildcard (*) fields,
which causes precision loss. Without precise reasoning of loop iterations (as done in an expensive shape analysis [33]), it is difficult to understand how many times a particular field reference appears in a recursive data structure.

In summary, our analysis makes it possible for the developer to spend an average of 13.7 seconds obtaining precise reference path information that would have otherwise taken several hours to profile (cf. Figure 1).

6.2 Usefulness Studies

To understand whether the reference paths are indeed useful to explain problems, we have manually checked all the reference paths produced by our analysis for 20 stale objects reported by the Sleigh memory leak detector and 20 racy objects reported by the Pacer race detector in each program. Our experience shows that the reported reference paths are useful to pinpoint root causes for all but a few trivial cases that can be easily understood without any context information. Figure 5 – 8 show an array of representative cases in which reference contexts provide significant benefit for problem diagnosis.

Memory Leak Diagnosis. Figure 5 shows an example in PMD — a static analysis based bug detector for Java. CCU reports that a stale object, which is created in method `getImage()` on line 415 of class `JavaCharStream` shown in Figure 5(a). The calling context reveals that the last use site of this object is on line 4023 of method `LabeledStatement()`, whose source code is shown in Figure 5(b). An inspection of the source code leads us to suspect that the stale object is cached in the image field of a `Token` object. However, there are many different kinds (subclasses) of `Token`. Developing a fix is impossible without knowing the particular `Token` type that unnecessarily references the stale object. This information is revealed explicitly by our reference path shown in Figure 5(c), which shows that only the image field of `GTToken` objects contains unnecessary references. They should be removed after the referenced strings are used.

Figure 6 shows an example in Bloat. The leaking object report is a `HashSet` object created in the constructor of class `GraphNode` and referenced by its `field preds`. As shown in Figure 6(a) and (b), the last use site of the object is in the iterator code where the set is retrieved from `field preds` in order to remove an object from it. The calling context shows that method `iterator()` is invoked when the dominance frontier is computed, which does not give us any useful information of why it is not used again after a node is removed. Our static analysis reports two important reference paths (Figure 6(c)). The first one shows that the leaking object, although created in `GraphNode`, is actually referenced by an object of `Block`, which is a subtype of `GraphNode`. This information is very valuable because `GraphNode` has more than 20 subclasses, each of which has a specific structure (e.g., with different numbers and types of predecessors and successors). Knowing the subtype `Block` immediately directs us to check its logic of adding/removing predecessors.

The second reference path shows us a more surprising piece of information — the leaking object is also referenced by an inner class `Graph$EdgeSet`. This class is created inside `Graph` to facilitate edge traversal. When an edge iterator is created for a node, the node’s predecessor set (i.e., the leaking object) is cached in its `Graph$EdgeSet`. Hence, just releasing

Figure 5. Memory leak example in PMD; each site is presented as \textit{"class.function(args) returnType:lineNum"}.

Figure 6. Memory leak example in Bloat.

Figure 7. Memory leak example in Lucene search.

Figure 8. Data race example in Xalan.
the leaking object from Block is not enough; we need to additionally clear the cache from Graph$EdgeSet.

Figure 7 shows another leak extracted from Lucene search. With the calling context information and the stale object’s allocation site in Figure 7(a), we can see that an object of type SegmentTermEnum is cloned and then caches the cloned object in a field of itself (termInfo is a field of object SegmentTermEnum) (in Figure 7(b)). By investigating the reference path in Figure 7(c), we understand that it is the ThreadLocal object that caches the original SegmentTermEnum object, which, in turn, caches its clone. The leak can be fixed by letting ThreadLocal object cache the clone instead of the original object. Finding where the original SegmentTermEnum object is referenced is nearly impossible by inspecting only the calling context — the store that writes it into the ThreadLocal object is executed during thread creation, which is in the DaCapo harness code that is not even part of Lucene.

Data Race Diagnosis. Figure 8 shows a data race example reported by Pacer [7]. The race occurs when the read statement at line 352 and the write statement at line 361 (shown in Figure 8(b)) are executed concurrently without any protection. Many other races reported by Racer are also w.r.t. the same field m_text_properties. It is easy to see that we need to protect this field, but knowing where this protection should be added is challenging. After seeing the reference path reported by our analysis (Figure 8(c)), it is immediately clear to us that m_text_properties is a field of a TransformerImpl object, which is referenced by an XPathContext object.

To safely process different XML files in different threads, we should let XPathContext reference a distinct TransformerImpl object for each thread, instead of sharing one single TransformerImpl object among threads. To fix the race, we can change the field m_owner in XPathContext to a hash map or array, each element of which stores a TransformerImpl object for a thread. Note that without this reference context, we would have focused only on the field m_text_properties and developed a naïve fix by synchronizing all accesses of this field. This has two problems: (1) it is clearly not as efficient as the synchronization-free approach mentioned above; and (2) TransformerImpl has many other fields that are not protected as well; only synchronizing m_text_properties is not sufficient.

We have also found many cases in Eclipse that are similar to this one — one single data processor (such as parser, tokenizer, etc.) object is used by multiple threads to process different data items. These cases are omitted due to space limitations. Pacer reports races w.r.t. multiple fields of the processor object. Without the reference information, one would have to develop many synchronizations to ensure that the one processor can be used safely by multiple threads. Inspecting the reference paths would quickly direct the developer’s attention to the classes higher than the processor class on the reference hierarchy (e.g., its direct and transitive owning classes), helping her understand how the processor is retrieved in different threads and determine in what class fixes can be added to achieve both safety and efficiency.

7 Related Work

Many dynamic analyses [7, 9, 19, 32, 34, 39, 42, 43, 54] need to profile either calling contexts or reference contexts. The CFL-reachability formulation [47] is first introduced for database query evaluation. Later work of Reps et al. [13, 26, 28, 30, 31] proposes to model realizable paths using a context-free language that treats method calls and returns as pairs of balanced parentheses.

CFL-reachability can be used to formulate many static analyses, such as polymorphic flow analysis [24], shape analysis [27], context-insensitive [38, 52] and context-sensitive [38, 46] points-to analysis, and information flow analysis [20]. The work in [16, 22] studies the connection between CFL-reachability and set-constraints, shows the connection between the two problems. Kodumal et al. [17] extend the set constraints to express analyses involving one context-free and any regular reachability properties.

CFL-reachability is also investigated in the context of recursive state machines [3], streaming XML [2], and pushdown languages [4]. Sridharan et al. define a CFL-reachability formulation to precisely model heap accesses, which results in demand-driven points-to analyses for Java [37, 38, 41]. Combining the CFL-reachability formulations of both heap accesses and interprocedural realizable paths, [37] proposes a context-sensitive analysis that achieves high precision by refining points-to relationships. Zheng and Rugina [53] propose a CFL-reachability-based formulation of demand-driven alias analysis for C. Xu et al. [45] propose the symbolic points-to graph (SPG) representation of the program. Recent work used CFL-reachability for library summarization [40] or specification inference [5]. Much work [10, 12, 49–51] has also been done to improve the efficiency of solving CFL-reachability.

8 Conclusion

The generality of CG-CFL is much beyond this particular application. It defines a general and sound framework for a class of interleaved language (A ∩ B) reachability problems. In particular, the framework can be used to answer queries of the form: if there are constraints over the strings of one language (A) and/or over the CFL paths, what should the strings of the other language (B) look like? Future work will further explore this formulation and develop new applications.

Acknowledgments

We thank Mooly Sagiv for shepherding the paper and the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under the grants CCF-1528133, CCF-1618158, CNS-1613023, and CNS-1703598, and by the Office of Naval Research under the grants N00014-14-1-0549, N00014-16-1-2913, and N00014-18-1-2037.