Verifying Correct Usage of Context-Free API Protocols

KOSTAS FERLES, The University of Texas at Austin, USA
JON STEPHENS, The University of Texas at Austin, USA
ISIL DILLIG, The University of Texas at Austin, USA

Several real-world libraries (e.g., reentrant locks, GUI frameworks, serialization libraries) require their clients to use the provided API in a manner that conforms to a context-free specification. Motivated by this observation, this paper describes a new technique for verifying the correct usage of context-free API protocols. The key idea underlying our technique is to over-approximate the program's feasible API call sequences using a context-free grammar (CFG) and then check language inclusion between this grammar and the specification. However, since this inclusion check may fail due to imprecision in the program's CFG abstraction, we propose a novel refinement technique to progressively improve the CFG. In particular, our method obtains counterexamples from CFG inclusion queries and uses them to introduce new non-terminals and productions to the grammar while still over-approximating the program's relevant behavior.

We have implemented the proposed algorithm in a tool called CFPChecker and evaluate it on 10 popular Java applications that use at least one API with a context-free specification. Our evaluation shows that CFPChecker is able to verify correct usage of the API in clients that use it correctly and produces counterexamples for those that do not. We also compare our method against three relevant baselines and demonstrate that CFPChecker enables verification of safety properties that are beyond the reach of existing tools.

CCS Concepts: • **Software and its engineering → Software verification**; • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: Context-Free API Protocols, Program Verification, Abstraction Refinement

ACM Reference Format:
Kostas Ferles, Jon Stephens, and Isil Dillig. 2021. Verifying Correct Usage of Context-Free API Protocols. *Proc. ACM Program. Lang.* 5, POPL, Article 17 (January 2021), 30 pages. https://doi.org/10.1145/3434298

1 INTRODUCTION

Over the last decade, there has been a flurry of research activity on checking the correct usage of APIs [Aldrich et al. 2009; Arzt et al. 2015; Bierhoff and Aldrich 2007; Bierhoff et al. 2009; Fink et al. 2008; Joshi and Sen 2008; Lam et al. 2004; Pradel et al. 2012a]. Despite significant advances in this area, almost all existing verification techniques focus on *typestate analysis* [Strom and Yemini 1986], which requires the API protocol to be expressible as a *regular language*. In reality, however, several APIs have context-free –rather than regular– specifications. For instance, almost all reentrant lock APIs require calls to `lock` to be balanced by a corresponding call to `unlock`. Similarly, many APIs provide functionality for saving and restoring internal state, and it is an error to call `restore` more times than the corresponding `save` function. As a final example, in APIs for structured document formats (e.g., JSON), the usage of the library needs to conform to the underlying context-free

Authors' addresses: Kostas Ferles, The University of Texas at Austin, USA, kferles@cs.utexas.edu; Jon Stephens, The University of Texas at Austin, USA, jon@cs.utexas.edu; Isil Dillig, The University of Texas at Austin, USA, isil@cs.utexas.edu.

2475-1421/2021/1-ART17
https://doi.org/10.1145/3434298

document specification. All of these examples are instances of context-free API protocols, and incorrect usage of such APIs typically results in run-time exceptions or resource leaks.

Motivated by this observation, prior research has developed *run-time* techniques for specifying context-free properties and monitoring them during program execution [d'Amorim and Havelund 2005; Jin et al. 2012; Martin et al. 2005; Meredith et al. 2010]. However, there has been very little (if any) work on *statically* verifying conformance between a program and a context-free API protocol. In this paper, we present a new verification technique that addresses this problem. In particular, given a specification expressed as a *parameterized* context-free grammar (CFG) $\mathcal{G}_S$ and a program $\mathcal{P}$ using that API, our method automatically checks whether or not $\mathcal{P}$ conforms to protocol $\mathcal{G}_S$. However, solving this problem introduces two key technical challenges that motivate the novel components of our solution: First, we need to prove that the program satisfies the API protocol for all, potentially infinite, relevant objects created by the input program. To address this challenge, we propose a novel program instrumentation that transforms the input program so it uses the same vocabulary as $\mathcal{G}_S$ and ensures that if the transformed program conforms to the API protocol so does the original. Second, because such APIs are often used in recursive procedures, it is important to reason precisely both about inter-procedural control flow as well as feasible API call sequences. Since *both* of these properties, namely matching call-and-return structure as well as the target API protocol, are context-free, standard program analysis techniques, such as CFL reachability [Reps et al. 1995] or visibly pushdown automata [Alur and Madhusudan 2004], do not address our problem. Instead, we reduce the context-free protocol verification problem to that of checking inclusion between two CFGs[1] and propose a *counterexample-guided abstraction refinement (CEGAR)* approach for checking whether *every* feasible execution of the program belongs to the grammar defined by the protocol (see Figure 1).

The heart of our technique consists of a novel abstraction mechanism that represents the input program $\mathcal{P}$ as a context-free grammar $\mathcal{G}_\mathcal{P}$, whose language $\mathcal{L}(\mathcal{G}_\mathcal{P})$ defines $\mathcal{P}$'s feasible API call sequences. The productions R of this grammar model relevant API calls as well as intra- and inter-procedural control-flow. For instance, a production such as $L_1 \rightarrow fL_2$ indicates that API method f is called at program location L_1 and that L_2 is a successor of L_1. In addition, productions precisely model inter-procedural control flow and enforce that every call statement must be matched by its corresponding return.

While the CFG extracted from the program is always *sound*, it may be *imprecise* due to data dependencies that are not captured by the current CFG productions. That is, if an API call sequence w is feasible in some program execution, then w is guaranteed to be in $\mathcal{L}(\mathcal{G}_\mathcal{P})$; however, the membership of w in $\mathcal{L}(\mathcal{G}_\mathcal{P})$ does not guarantee the feasibility of the corresponding API call sequence. Our verification approach deals with this potential imprecision by using a novel abstraction refinement technique that iteratively improves the program's CFG abstraction until the property can be either refuted or verified.

In more detail, our approach works as follows: First, given context-free protocol $\mathcal{G}_S$ and current program abstraction $\mathcal{G}_\mathcal{P}$, we query whether there exists a word w that is in $\mathcal{G}_\mathcal{P}$ but not $\mathcal{G}_S$. If not, then the algorithm terminates with a proof of correctness. Otherwise, our method reconstructs the corresponding program path π associated with w and checks its feasibility using an SMT solver. If π is indeed feasible, then so is the call sequence w, and our method terminates with a real counterexample. Otherwise, w must be a *spurious* counterexample caused by imprecision in the CFG. In this case, our algorithm refines the CFG abstraction by computing a proof of infeasibility of π in the form of a *nested sequence interpolant* [Heizmann et al. 2010]. Similar to many other software

[1]While inclusion checking between two CFGs is undecidable, many problems of practical interest can be solved by existing tools.

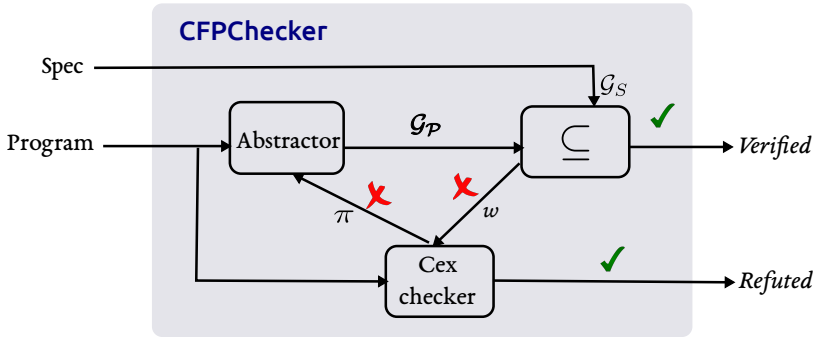


Fig. 1. Overview of verification approach

model checkers, the interpolant drives the refinement process inside the CEGAR loop; however, *unlike* other techniques, our approach uses the interpolant to figure out which new non-terminals and productions to add to the grammar. In essence, these new non-terminals correspond to "clones" of existing program locations and allow us to selectively introduce both intra- and inter-procedural path-sensitivity to our CFG-based program abstraction.

We have implemented our proposed verification algorithm in a prototype called CFPChecker for Java programs and evaluated it on 10 widely-used clients of 5 popular APIs with context-free specifications. Our evaluation demonstrates that CFPChecker is able to verify correct usage of the API in clients that use it correctly and produces counterexamples for those that do not. We also implement and evaluate three baselines that reduce the problem to assertion checking and then discharge these assertions using existing tools. Our experiments demonstrate that CFPChecker is practical enough to successfully analyze real-world Java applications and that it enables the verification of safety properties that are beyond the reach of existing tools.

In summary, this paper makes the following contributions:

- We propose a novel CEGAR-based verification algorithm for verifying correct usage of context-free API protocols.
- We describe a new CFG-based program abstraction that over-approximates feasible API call sequences.
- We propose a new refinement method that *selectively and modularly* adds path-sensitivity to the program abstraction by introducing new non-terminals and productions.
- We evaluate our method on widely-used clients of popular Java APIs with context-free specifications and demonstrate that our proposed approach is applicable to real-world software verification tasks.

2 MOTIVATING EXAMPLE

In this section, we give a high level overview of our approach through a simple motivating example. Consider a re-entrant lock API that requires every call to `lock` on some object `o` to be matched by the same number of calls to `unlock` on `o`. This property is context-free but not regular because it requires "counting" the number of calls to `lock` and `unlock`. In our framework, the user can specify this property using the following parametrized context-free grammar $\mathcal{G}_S$:

$$S \rightarrow \epsilon \mid \$1.lock()\ S\ \$1.unlock()\ S \tag{1}$$

```
1  void foo(Lock l){
2    if (*) {
3      acquire(l);
4      foo(l);
5      release(l);
6    }
7  }
8
9  void acquire(Lock l1){
10   l1.lock();
11 }
12
13 void release(Lock l2){
14   l2.unlock();
15 }
```

(a) Original Program

```
1  static Lock $1 = *;
2
3  void foo(Lock l){
4    if (*) {
5      acquire(l);
6      foo(l);
7      release(l);
8    }
9  }
10
11 void acquire(Lock l1){
12   if (l1 == $1)
13     $1.lock();
14 }
15
16 void release(Lock l2){
17   if (l2 == $1)
18     $1.unlock();
19 }
```

(b) Transformed Program

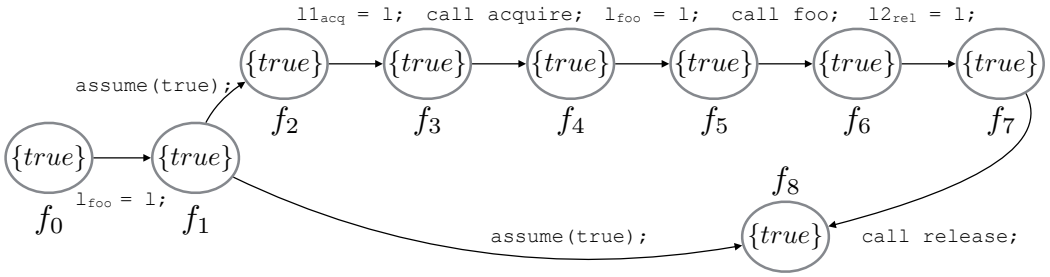
Fig. 2. Motivating Example

This CFG is parametrized in the sense that it uses a "wildcard" symbol $1 that matches any object of type `Lock`. Thus, the specification requires that, for *every* object *o*, each call `o.lock()` must be matched by a call to `o.unlock()`.

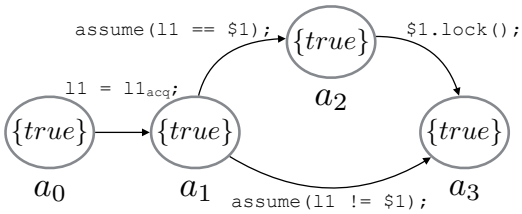
To illustrate our technique, Figure 2(a) shows a very simple client of this `Lock` API. Here, `foo` is a recursive procedure that calls `l.lock` before every recursive call to `foo` and calls `l.unlock` afterwards. Since the receiver object is the same before and after the call, the specification from Equation 1 is satisfied. In the remainder of this section, we explain how our technique verifies correct usage of the `Lock` API in this example.

The first step in our technique is to automatically instrument the program from Figure 2(a) so that API calls in the program involve the same wildcard symbol `$1` used in the specification. The instrumented version is shown in Figure 2(b), which uses a new global variable called `$1` (i.e., the wildcard symbol in the grammar) and replaces every call to `x.lock()` (resp. `x.unlock()`) with the conditional invocation `if(x = $1) $1.lock()` (resp. `if(x = $1) $1.unlock()`). Intuitively, the goal of this instrumentation is two-fold: First, it ensures that the CFG abstraction of the program uses the same "vocabulary" (i.e., terminals) as the specification CFG. Second, it deals with challenges that arise from potential aliasing between pointers.

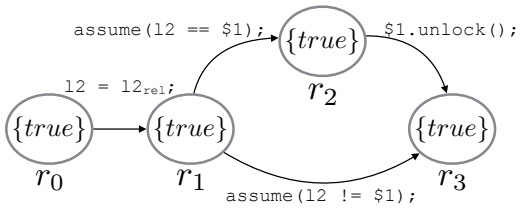
In the next step, our method extracts a context-free grammar that over-approximates the relevant API call behavior of the program. Towards this goal, we represent the program as a mapping from each function to a *predicated control-flow automaton* (*PCFA*) that will be iteratively refined as the algorithm progresses. At a high level, a PCFA captures control-flow within a method while also maintaining a mapping from program locations to a set of logical predicates. For example, Figure 3 shows the initial PCFAs for Figure 2(b): here, nodes correspond to program locations, and edges correspond to transitions. Observe that the PCFAs from Figure 3 contain a *single* node for each program location; hence, these PCFAs look like standard *control flow automata (CFA)*



(a) Initial *PCFA* for foo.



(b) Initial *PCFA* for acquire.



(c) Initial *PCFA* for release.

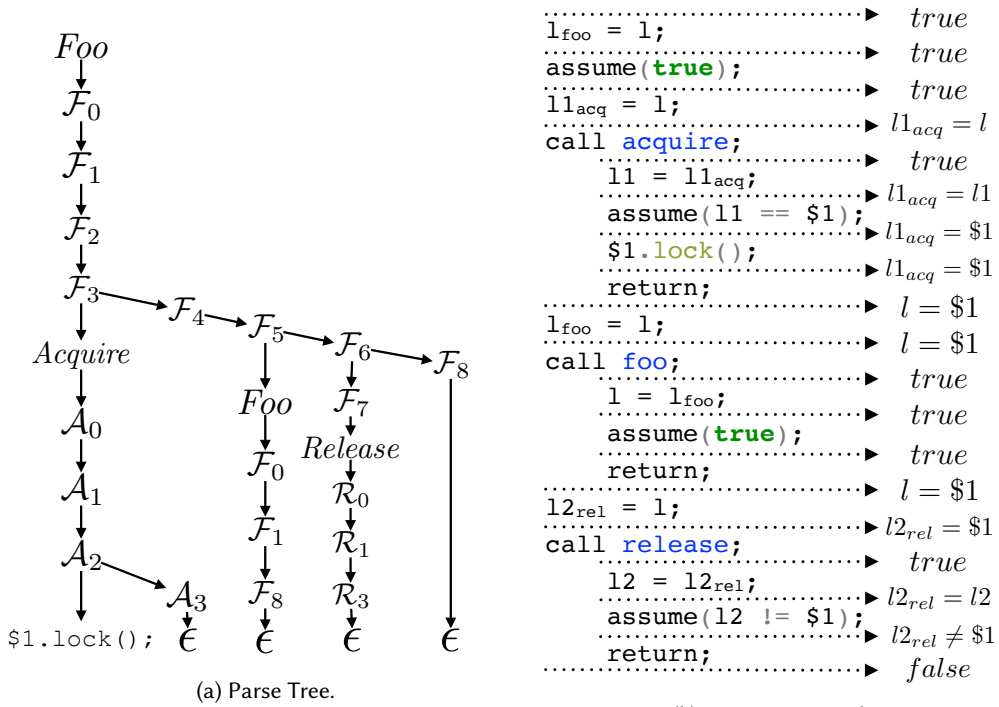
Fig. 3. Initial *PCFAs* for input program. The PCFAs contain additional formal-to-actual assignments.

$$
\begin{aligned}
\mathit{Foo} &\rightarrow \mathcal{F}_0\\
\mathcal{F}_0 &\rightarrow \mathcal{F}_1\\
\mathcal{F}_1 &\rightarrow \mathcal{F}_2 \mid \mathcal{F}_8\\
\mathcal{F}_2 &\rightarrow \mathcal{F}_3\\
\mathcal{F}_3 &\rightarrow \mathit{Acquire}\ \mathcal{F}_4\\
\mathcal{F}_4 &\rightarrow \mathcal{F}_5\\
\mathcal{F}_5 &\rightarrow \mathit{Foo}\ \mathcal{F}_6\\
\mathcal{F}_6 &\rightarrow \mathcal{F}_7\\
\mathcal{F}_7 &\rightarrow \mathit{Release}\ \mathcal{F}_8\\
\mathcal{F}_8 &\rightarrow \epsilon
\end{aligned}
\qquad
\begin{aligned}
\mathit{Acquire} &\rightarrow \mathcal{A}_0\\
\mathcal{A}_0 &\rightarrow \mathcal{A}_1\\
\mathcal{A}_1 &\rightarrow \mathcal{A}_2 \mid \mathcal{A}_3\\
\mathcal{A}_2 &\rightarrow \texttt{\$1.lock()}\ \mathcal{A}_3\\
\mathcal{A}_3 &\rightarrow \epsilon
\end{aligned}
\qquad
\begin{aligned}
\mathit{Release} &\rightarrow \mathcal{R}_0\\
\mathcal{R}_0 &\rightarrow \mathcal{R}_1\\
\mathcal{R}_1 &\rightarrow \mathcal{R}_2 \mid \mathcal{R}_3\\
\mathcal{R}_2 &\rightarrow \texttt{\$1.unlock()}\ \mathcal{R}_3\\
\mathcal{R}_3 &\rightarrow \epsilon
\end{aligned}
$$

Fig. 4. Initial context-free grammar.

used in software model checking [Heizmann et al. 2010; Reps et al. 1995]. However, the PCFA representation diverges from a standard CFA as the algorithm proceeds. In particular, the PCFA can contain multiple nodes for the same program location and allows our method to selectively introduce path-sensitivity to the program abstraction.

Given these initial PCFAs, our method programmatically extracts from them a context-free grammar over-approximating the program's feasible API call sequences. In particular, Figure 4 shows the initial CFG abstraction for our example. Here, non-terminals (e.g., $\mathcal{F}_1, \mathcal{A}_2$) correspond to nodes (e.g., f_1, a_2) in the PCFAs, and terminals (e.g., `$1.lock()`) denote API calls. Additionally, there is one non-terminal symbol (e.g., *Foo, Acquire*) for each method. The productions in the CFG are obtained directly from the PCFA by ignoring all statements that are not function calls:



(a) Parse Tree.

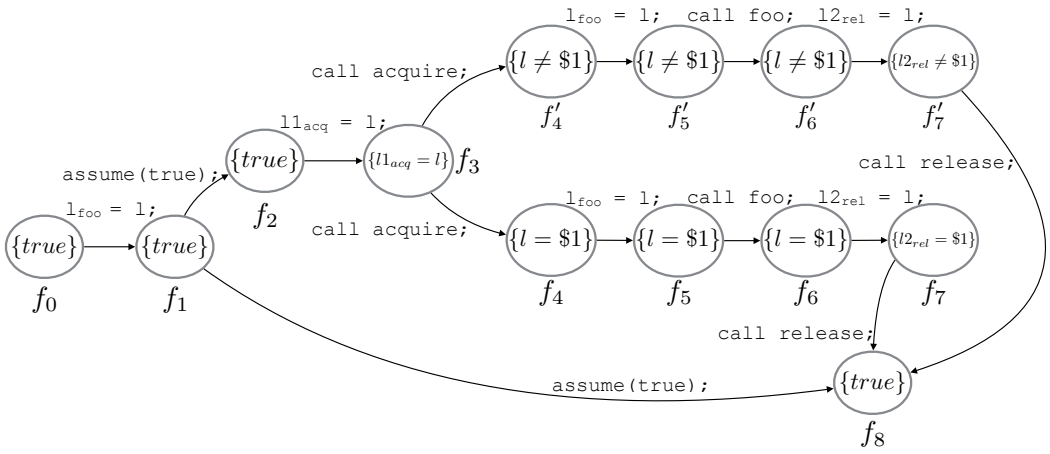
(b) Trace & Interpolants.

Fig. 5. Tree and Trace for Counterexample `$1.lock()`.

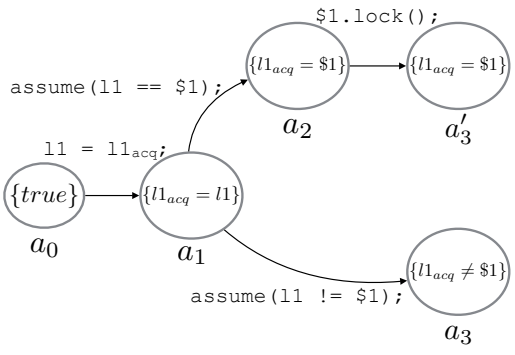
For example, the production $\mathcal{A}_2 \rightarrow$ `$1.lock()` $\mathcal{A}_3$ comes from the PCFA edge from a_2 to a_3. In addition, the CFG productions faithfully and precisely model inter-procedural control flow. For instance, the production $\mathcal{F}_3 \rightarrow$ *Acquire* $\mathcal{F}_4$ models the call from *Foo* to *Acquire* and $\mathcal{A}_3 \rightarrow \epsilon$ models its corresponding return.

Next, our method checks inclusion between the grammar $\mathcal{G}_\mathcal{P}$ extracted from the program and API protocol $\mathcal{G}_S$. While this problem is, in general, undecidable, we have found the resulting CFG inclusion checking problems to be amenable to automation by modern tools. Going back to our running example, the language of $\mathcal{G}_\mathcal{P}$ from Figure 4 is *not* a subset of the language of $\mathcal{G}_S$ — for example, the word `$1.lock()` can be generated using $\mathcal{G}_\mathcal{P}$ but not $\mathcal{G}_S$. This means that either the program actually misuses the API or the current abstraction is imprecise. In order to determine which one, our method maps the word `$1.lock()` to an execution path of the program. Towards this goal, we first obtain the parse tree from Figure 5a that shows how `$1.lock()` can be derived from $\mathcal{G}_\mathcal{P}$. This derivation corresponds precisely to the program path, shown in Figure 5b. Furthermore, observe that this path goes through the "then" branch of the `if` statement in method `acquire` and the "else" branch in method `release`. However, this path is clearly infeasible, so we need to refine $\mathcal{G}_\mathcal{P}$ to eliminate the spurious derivation.

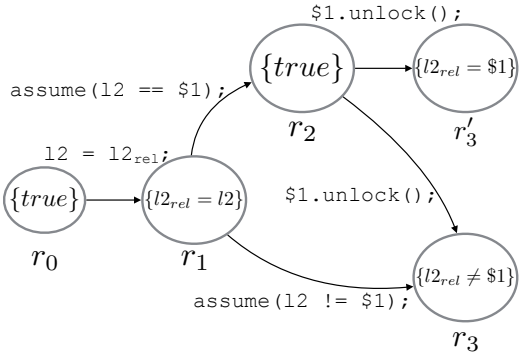
Our method refines the program's CFG abstraction by adding new non-terminals and productions to the grammar. Towards this goal, we first refine the PCFA abstraction by selectively cloning some program locations, with the goal of introducing path-sensitivity where needed. The cloning of PCFA nodes is driven by an interpolation engine that computes a *sequence of nested interpolants* [Heizmann et al. 2010]. In particular, the right-hand side of Figure 5b shows the interpolants computed for



(a) Refined *PCFA* for foo.



(b) Refined *PCFA* for acquire.



(c) Refined *PCFA* for release.

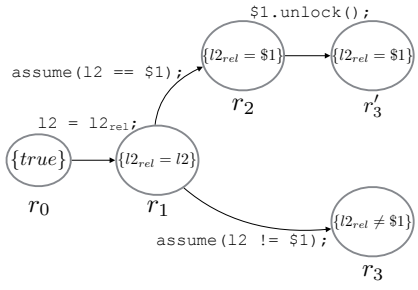
Fig. 6. Refined *PCFAs* for input program.

each program location for our running example. Intuitively, "tracking" these predicates at the corresponding program location would allow us to remove the spurious trace. Thus, in the next iteration, we generate the new PCFAs shown in Figure 6 by cloning all PCFA nodes that correspond to program locations in the counterexample. Observe that the refined PCFAs contain multiple nodes (e.g., f_4, f_4') for the same program location, and the predicates in the PCFA correspond to those that appear in the interpolant. For instance, even though nodes r_3, r_3' both represent the same program location, one is annotated with predicate $l2_{rel} \neq \$1$, whereas r_3' is annotated with $l2_{rel} = \$1$. Furthermore, the refined PCFA contains an edge between two nodes iff the semantics of the statement labeling that edge are consistent with the annotations of the source and target nodes. For instance, there is an edge from node a_1 to a_3 but not from a_1 to a_3' because the predicates $l1_{acq} = l1$, $l1_{acq} = \$1$ labeling a_1 and a_3' are inconsistent with the statement `assume(l1 != $1)`.

Given this new PCFA representation, our verification algorithm extracts the refined grammar $\mathcal{G}'_{\mathcal{P}}$ shown in Figure 7. As before, we construct the grammar based on PCFA edges; however, note that there are two different sets of grammar rules for each of the methods `acquire` and `release`. In general, for a given function f, our technique introduces as many non-terminals for f as there are

$$
\begin{array}{lll}
\mathit{Foo} \rightarrow \mathcal{F}_0 & & \\
\mathcal{F}_0 \rightarrow \mathcal{F}_1 & & \mathit{Release}_{\phi_3} \rightarrow \mathcal{R}_{0,\phi_3} \\
\mathcal{F}_1 \rightarrow \mathcal{F}_2 \mid \mathcal{F}_8 & \mathit{Acquire}_{\phi_1} \rightarrow \mathcal{A}_{0,\phi_1} & \mathcal{R}_{0,\phi_3} \rightarrow \mathcal{R}_{1,\phi_3} \\
\mathcal{F}_2 \rightarrow \mathcal{F}_3 & \mathcal{A}_{0,\phi_1} \rightarrow \mathcal{A}_{1,\phi_1} & \mathcal{R}_{1,\phi_3} \rightarrow \mathcal{R}_{2,\phi_3} \\
\mathcal{F}_3 \rightarrow \mathit{Acquire}_{\phi_1}\ \mathcal{F}_4 & \mathcal{A}_{1,\phi_1} \rightarrow \mathcal{A}_{2,\phi_1} & \mathcal{R}_{2,\phi_3} \rightarrow \texttt{\$1.unlock()}\ \mathcal{R}'_{3,\phi_3} \\
\quad \mid \mathit{Acquire}_{\phi_2}\ \mathcal{F}'_4 & \mathcal{A}_{2,\phi_1} \rightarrow \texttt{\$1.lock()}\ \mathcal{A}'_{3,\phi_1} & \mathcal{R}'_{3,\phi_3} \rightarrow \epsilon \\
\mathcal{F}_4 \rightarrow \mathcal{F}_5 & \mathcal{A}'_{3,\phi_1} \rightarrow \epsilon & \\
\mathcal{F}'_4 \rightarrow \mathcal{F}'_5 & & \\
\mathcal{F}_5 \rightarrow \mathit{Foo}\ \mathcal{F}_6 & & \mathit{Release}_{\phi_4} \rightarrow \mathcal{R}_{0,\phi_4} \\
\mathcal{F}'_5 \rightarrow \mathit{Foo}\ \mathcal{F}'_6 & \mathit{Acquire}_{\phi_2} \rightarrow \mathcal{A}_{0,\phi_2} & \mathcal{R}_{0,\phi_4} \rightarrow \mathcal{R}_{1,\phi_4} \\
\mathcal{F}_6 \rightarrow \mathcal{F}_7 & \mathcal{A}_{0,\phi_2} \rightarrow \mathcal{A}_{1,\phi_2} & \mathcal{R}_{1,\phi_4} \rightarrow \mathcal{R}_{2,\phi_4} \\
\mathcal{F}'_6 \rightarrow \mathcal{F}'_7 & \mathcal{A}_{1,\phi_2} \rightarrow \mathcal{A}_{3,\phi_2} & \mathcal{R}_{1,\phi_4} \rightarrow \mathcal{R}_{3,\phi_4} \\
\mathcal{F}_7 \rightarrow \mathit{Release}_{\phi_3}\ \mathcal{F}_8 & \mathcal{A}_{3,\phi_2} \rightarrow \epsilon & \mathcal{R}_{2,\phi_4} \rightarrow \texttt{\$1.unlock()}\ \mathcal{R}_{3,\phi_4} \\
\mathcal{F}'_7 \rightarrow \mathit{Rlease}_{\phi_4}\ \mathcal{F}_8 & & \mathcal{R}_{3,\phi_4} \rightarrow \epsilon \\
\mathcal{F}_8 \rightarrow \epsilon & &
\end{array}
$$

Fig. 7. Refined CFG, where $\phi_1 = \{l1_{acq} = \$1\}$, $\phi_2 = \{l1_{acq} \neq \$1\}$, $\phi_3 = \{l2_{rel} = \$1\}$, and $\phi_4 = \{l2_{rel} \neq \$1\}$.



(a) Second refined PCFA for method `release`.

$$
\begin{array}{l}
\mathit{Release}_{\phi_3} \rightarrow \mathcal{R}_{0,\phi_3} \\
\mathcal{R}_{0,\phi_3} \rightarrow \mathcal{R}_{1,\phi_3} \\
\mathcal{R}_{1,\phi_3} \rightarrow \mathcal{R}_{2,\phi_3} \\
\mathcal{R}_{2,\phi_3} \rightarrow \texttt{\$1.unlock()}\ \mathcal{R}'_{3,\phi_3} \\
\mathcal{R}'_{3,\phi_3} \rightarrow \epsilon
\end{array}
$$

$$
\begin{array}{l}
\mathit{Release}_{\phi_4} \rightarrow \mathcal{R}_{0,\phi_4} \\
\mathcal{R}_{0,\phi_4} \rightarrow \mathcal{R}_{1,\phi_4} \\
\mathcal{R}_{1,\phi_4} \rightarrow \mathcal{R}_{3,\phi_4} \\
\mathcal{R}_{3,\phi_4} \rightarrow \epsilon
\end{array}
$$

(b) Refined grammar for *Release*.

Fig. 8. Refined PCFA and grammar for method `release` (second iteration).

PCFA nodes for f's exit location. This strategy allows our verification algorithm to lazily perform "method cloning", thereby introducing *inter-procedural* path-sensitivity where needed. For instance, observe that there are two non-terminals ($\mathit{Acquire}_{\phi_1}$, $\mathit{Acquire}_{\phi_2}$) representing `acquire` in Figure 7, and predicates ϕ_1, ϕ_2 correspond to the predicates $l1_{acq} = \$1$, $l1_{acq} \neq \$1$ labeling nodes a_3 and a'_3 in the PCFA from Figure 6(b). Furthermore, observe that there are two different sets of grammars for $\mathit{Acquire}_{\phi_1}$ and $\mathit{Acquire}_{\phi_2}$, and each grammar is generated by looking at the portion of the PCFA that is backwards reachable from the corresponding exit node. For example, there is no production $\mathcal{A}_{1,\phi_2} \rightarrow \mathcal{A}_{2,\phi_2}$ in Figure 7 because node a_2 is not backwards reachable from the exit node labeled with ϕ_2 in Figure 6(b).

Class C ::= `class` C { fld^* m^* }
Field fld ::= $f : \tau = e$ | `static` $f : \tau = e$
Method m ::= `void` $m(\vec{v})$ {s^*;}
Stmt s ::= $skip$ | $s_1; s_2$ | $v := e$ | $v.f := e$ | `assume`(p) | `if` (p) {s_1} `else` {s_2} | $v :=$ `new` C
| `call` $v.m(\vec{v})$ | `api_call` $v.m(\vec{v})$
Expr e ::= v | $v.f$ | c | $*$ | $e_1 \ominus e_2, \ominus \in \{+, -, \times\}$
Pred p ::= e | $\neg p$ | $p_1 \wedge p_2$ | $p_1 \vee p_2$ | $e_1 \oplus e_2,\ \oplus \in \{<, >, =\}$

Fig. 9. Input Language.

In the second iteration, our algorithm again checks inclusion between the two grammars, namely $\mathcal{G}'_{\mathcal{P}}$ and $\mathcal{G}_S$. This time, $\mathcal{G}'_{\mathcal{P}}$ is still not contained in $\mathcal{G}_S$, and the new counterexample is `$1.unlock()`, whose derivation corresponds to a program path that goes through the "else" branch in `acquire` and "then" branch in `release`. In this case, the culprit is the PCFA edge between nodes r_2 and r_3 in method `release` (Figure 6c), which can again be eliminated by computing nested interpolants and cloning node r_2.

In the next and final iteration, our algorithm can now prove that the language defined by the program's CFG is indeed a subset of the specification $\mathcal{G}_S$, and the algorithm terminates with a proof of correctness. The final abstraction is identical to the one from previous iteration except for method `release` whose final PCFA and context-free grammar are shown in Figure 8.

3 PROBLEM STATEMENT

In this section, we introduce context-free API protocols and formally define our problem in the context of a simple object-oriented programming language.

3.1 Input Language

Figure 9 presents the programming language used for our formalization. In this language, a class consists of a set of field declarations followed by a set of method definitions. Fields can be either object-specific (declared as $f : \tau$) or `static`, meaning they are shared between all instances of the class. Statements include standard constructs like assignment, load, store, etc. We differentiate between two kinds of call statements, namely `call` which is a call to a regular method defined in the same program and `api_call` which invokes a method defined by a third-party API. We assume that the source code of third-party libraries are not available for analysis; thus, we require any side effects of API calls to be modeled using stub methods. In particular, we assume that each call to an API method `foo` in the original program has been replaced by a stub `foo_stub` that invokes `foo` and captures its side effects via assignment. *Thus, in the remainder of the paper, we assume, without loss of generality, that API calls have no side effects on program state.*

For the purposes of this paper, a program state σ is a mapping from program variables (V) and field references ($V \times F$) to an integer value. We use the notation $\langle s, \sigma \rangle \Downarrow \sigma'$ to indicate that σ' is the resulting state after executing statement s on program state σ. Furthermore, we use $sp(s, P)$ to denote the strongest postcondition of statement s with respect to the first-order logic formula P. A program trace, $\tau = \langle s_1, \sigma_1 \rangle, \langle s_2, \sigma_2 \rangle, ..., \langle s_n, \sigma_n \rangle$, is a sequence of (statement, program state) pairs such that $\langle s_i, \sigma_i \rangle \Downarrow \sigma_{i+1}$.[2] Given a program $\mathcal{P}$, we write $Traces(\mathcal{P})$ to denote the (infinite) set of traces that can arise during executions of $\mathcal{P}$.

[2]We assume that program traces are in SSA form. That is, each re-definition of a program variable is assigned a unique name within the trace.

3.2 Context-Free API Protocols

We express API protocols using a (parametrized) context-free grammar $\mathcal{G}_S = (T, N, R, S)$ where each terminal $t \in T$ is of the form "`api_call $i_1.m($i_2, ..., $i_p)`", $n \in N$ is a non-terminal, R is a set of productions, and S is the start symbol. Given grammar $\mathcal{G}_S$, we write T_m to denote the subset of terminals involving a call to method m. As mentioned in Section 2, each $\$i_j$ is a so-called *wildcard* that can match any value of the appropriate type. To omit explicit type declarations, we assume the existence of a typing oracle Γ that returns the type of a wildcard w, and, as standard, we use the notation $\Gamma \vdash w : \tau$ to indicate that w is of type τ. We also define a function to extract all wildcard symbols that appear in the grammar:

Definition 3.1. **(Wildcard extractor, $\mathcal{W}$)** Given a context-free protocol $\mathcal{G}_S = (T, N, R, S)$, we write $\mathcal{W}(\mathcal{G}_S)$ to denote the set of all wildcard symbols that appear in $\mathcal{G}_S$.

3.3 Semantic Conformance to API Protocol

Intuitively, a program $\mathcal{P}$ conforms to a parametrized CFG specification $\mathcal{G}_S$ if it satisfies the spec for every possible instantiation of the wildcards in $\mathcal{G}_S$. To make this statement more precise, we first introduce the notion of an *instantiated API protocol*:

Definition 3.2. **(Instantiated spec)** Given an API specification $\mathcal{G}_S$, we say that $\hat{\mathcal{G}}$ is an instantiation of $\mathcal{G}_S$, written $\hat{\mathcal{G}} \in \mathit{Inst}(\mathcal{G}_S)$, if it can be obtained from $\mathcal{G}_S$ by substituting every wildcard symbol $w_i \in \mathcal{W}(\mathcal{G}_S)$ with a concrete value of the appropriate type.

Next, to determine if a program trace τ conforms to an instantiated specification $\hat{\mathcal{G}}$, we will check "inclusion" of the trace in the language defined by $\hat{\mathcal{G}}$. To this end, we convert the trace to a word over the terminal symbols in $\hat{\mathcal{G}}$ using the following *TraceToWord* function:

Definition 3.3. **(Trace-to-Word)** Let τ be a trace and let $\hat{\mathcal{G}} = (T, N, R, S)$ be an (instantiated) API protocol. We define $\mathit{TraceToWord}(\tau, \hat{\mathcal{G}})$ as follows[3]:

$$\mathit{TraceToWord}(\tau, \hat{\mathcal{G}}) = [s' \mid s' \in T, \langle s,\ \sigma\rangle \in \tau,\ s' = s[\sigma(\vec{v})/\vec{v}],\ \vec{v} = \mathsf{Vars}(s)]$$

Example 3.4. Consider the following trace τ:

$$\tau = \langle \texttt{l1 = new Lock}, \sigma_1\rangle, \langle \texttt{l1.lock()}, \sigma_2\rangle, \langle \texttt{l1.unlock()}, \sigma_3\rangle,$$
$$\langle \texttt{l2 = new Lock}, \sigma_4\rangle, \langle \texttt{l2.lock()}, \sigma_5\rangle, \langle \texttt{l2.unlock()}, \sigma_6\rangle$$

and suppose that o_1, o_2 refer to the addresses of the first and second allocated `Lock` objects respectively. Now, consider the following instantiated spec $\hat{\mathcal{G}}$:

$$\hat{\mathcal{G}} \ = \ S \ \rightarrow \ \epsilon \mid o_1.\mathit{lock}()\ S\ o_1.\mathit{unlock}()\ S$$

Then, we have:

$$\mathit{TraceToWord}(\tau, \hat{\mathcal{G}}) = [o_1\texttt{.lock()}, o_1\texttt{.unlock()}]$$

Observe that the generated word "ignores" all statements other than API calls (e.g., `new Lock`). Furthermore, since variable `l2` has value o_2 rather than o_1, the last two `lock`/`unlock` statements in the trace are also not included in the result.

[3]We use the notation $[s \mid \ldots]$ to describe a filter operation on the input trace. The output preserves the relative order of statements in the input trace.

$$(API)\quad \frac{T_m = \{t_1, ..., t_k\}\ \ g_i = guard(t_i, s) \quad s' = \texttt{if}\ (g_1)\ t_1\ \ldots\ \texttt{else if}(g_k)\ t_k}{\Gamma, \mathcal{G}_S \vdash s = \texttt{api\_call}\ v.m(\vec{v}) \hookrightarrow s'}$$

$$(Seq)\quad \frac{\Gamma, \mathcal{G}_S \vdash s_1 \hookrightarrow s_1' \quad \Gamma, \mathcal{G}_S \vdash s_2 \hookrightarrow s_2'}{\Gamma, \mathcal{G}_S \vdash s_1; s_2 \hookrightarrow s_1'; s_2'}$$

$$(If)\quad \frac{\Gamma, \mathcal{G}_S \vdash s_1 \hookrightarrow s_1' \quad \Gamma, \mathcal{G}_S \vdash s_2 \hookrightarrow s_2'}{\Gamma, \mathcal{G}_S \vdash \texttt{if}(p)\ \{s_1\}\ \texttt{else}\{s_2\} \hookrightarrow \texttt{if}(p)\ \{s_1'\}\ \texttt{else}\ \{s_2'\}}$$

$$(Method)\quad \frac{\Gamma, \mathcal{G}_S \vdash s \hookrightarrow s'}{\Gamma, \mathcal{G}_S \vdash \texttt{void}\ m(\vec{v})\{s\} \hookrightarrow \texttt{void}\ m(\vec{v})\{s'\}}$$

$$(Class)\quad \frac{w_i \in \mathcal{W}(\mathcal{G}_S) \quad \Gamma \vdash w_i : \tau_i \quad f_i' = \texttt{static}\ w_i : \tau_i = * \quad \Gamma, \mathcal{G}_S \vdash m_i \hookrightarrow m_i'}{\Gamma, \mathcal{G}_S \vdash \texttt{class C \{}\ f_1 \ldots f_n\ m_1 \ldots m_k\ \texttt{\}} \hookrightarrow \texttt{class C \{}\ f_1 \ldots f_n\ f_1' \ldots f_j'\ m_1' \ldots m_k'\ \texttt{\}}}$$

Fig. 10. Rules for instrumenting program $\mathcal{P}$ for a given specification $\mathcal{G}_S = (T, N, R, S)$. For statements that are not shown, we have $\Gamma, \mathcal{G}_S \vdash s \hookrightarrow s$, and the definition of *guard* function is inlined in text.

Definition 3.5. **(Semantic conformance)** Given a program $\mathcal{P}$ and a context-free API protocol $\mathcal{G}_S$, $\mathcal{P}$ semantically conforms to $\mathcal{G}_S$ if and only if the following holds:

$$\forall \tau \in \mathit{Traces}(\mathcal{P}). \forall \hat{\mathcal{G}} \in \mathit{Inst}(\mathcal{G}_S).\ \mathit{TraceToWord}(\tau, \hat{\mathcal{G}}) \in \mathcal{L}(\hat{\mathcal{G}}) \tag{2}$$

In other words, a program $\mathcal{P}$ satisfies $\mathcal{G}_S$ if it satisfies the protocol for all possible instantiations of the wildcards in $\mathcal{G}_S$ for every program trace.

4 PROGRAM INSTRUMENTATION

In the previous section, we defined conformance of a program to an API protocol in terms of all possible program traces and all possible instantiations of the wildcard symbols. While this strategy allows us to formally state the problem, it does not lend itself to a verification algorithm since there are infinitely many possible instantiations of the wildcard symbols as well as infinitely many program traces. Thus, rather than checking the containment of each trace in all possible instantiations of the parametrized CFG, our strategy is to instead generate a CFG encoding all possible traces of the program as well as all possible instantiations of the wildcard symbols and then check inclusion between this CFG and the specification grammar. Towards this goal, we first *instrument* the program with new fields that are initialized non-deterministically and that can be used to capture all possible values of the wildcards in the specification. In addition, our instrumentation deals with challenges that arise from potential aliasing between different arguments to API calls.

In more detail, Figure 10 describes our program instrumentation using judgments of the form $\Gamma, \mathcal{G}_S \vdash s \hookrightarrow s'$, where s' corresponds to the transformed version of s.

Class. The top-level rule labeled "*Class*" introduces a static field for every wildcard symbol that appears in $\mathcal{G}_S$ and initializes it to a non-deterministic value. It also instruments each method within this class.

Method, Seq, If. These three rules reconstruct the statement after recursively transforming the statements nested inside them.

API. This rule is the core of our program instrumentation and ensures that each terminal symbol in the specification grammar has a (syntactically) corresponding API call statement while being semantically equivalent to the original API call. As shown in Figure 10, this rule transforms an API call s to library method m to an `if-then-else` statement. Specifically, the rule iterates over all the terminals $t_k \in T_m$ in $\mathcal{G}_S$ and generates an `if` statement for each terminal t_k conditioned upon the wildcard symbols matching the variables used in s. To achieve this goal, we make use of an auxiliary *guard* function defined as follows:

$$guard(t_k, s) = \bigwedge_j \$i_{kj} = v_j$$

Here, $\vec{\$i_k}$ is the sequence of wildcards used in t_k and $\vec{v}$ is the sequence of variables used in s. Thus, given an API call s and a set of terminals T_m, we generate the following code:

$$\texttt{if}(\$i_{11} = v_1 \ \wedge \ \ldots \ \wedge \ \$i_{1n} = v_n) \ \{ \ t_1 \ \}$$
$$\ldots$$
$$\texttt{else if}(\$i_{k1} = v_1 \ \wedge \ \ldots \ \wedge \ \$i_{kn} = v_n) \ \{ \ t_k \ \}$$

Hence, our instrumentation ensures that API calls syntactically use the wildcard symbols in the grammar while preserving program behavior relevant to the specification.

The following theorem states the correctness of our instrumentation:

Theorem 4.1. *Let $\mathcal{P}$ be a program and $\mathcal{G}_S$ a context-free API protocol. If we have $\Gamma, \mathcal{G}_S \vdash \mathcal{P} \hookrightarrow \mathcal{P}'$ and $\mathcal{P}'$ semantically conforms $\mathcal{G}_S$, then so does $\mathcal{P}$.*

Proof. The proofs of all theorems can be found in the extended version of the paper [Ferles et al. 2020]. □

Observe that the above theorem only states the soundness, but not completeness, of our program instrumentation. Completeness does not hold for arbitrary parametrized CFGs. For example, consider the API protocol: $\mathcal{G}_S \rightarrow \$1.f() \ \$2.g()$, where $1 and $2 have different types, and the code fragment "`v1.f() v2.g()`". This fragment clearly conforms to the API protocol, however, our instrumentation would produce the following output:

```
$1 = *; $2 = *;
if (v1 == $1) $1.f();
if (v2 == $2) $2.g();
```

The instrumented program does not satisfy the API protocol because it generates the words "$1.f()" and "$2.g()" that do not belong in $\mathcal{L}(\mathcal{G}_S)$. Such protocols typically do not occur in practice because such examples refer to relationships between methods defined in different classes, so this is no longer a protocol for a single API.

However, completeness *does* hold if all terminals in the grammar use the same set of wildcards. In practice, every API protocol we have encountered conforms to this restriction.

5 VERIFICATION ALGORITHM

Our verification algorithm takes as input a program that has been instrumented as described in Section 4. The main idea underlying the algorithm is to extract a context-free grammar from the instrumented program and iteratively refine this CFG abstraction until the property is either refuted

or verified. Since our algorithm operates over *predicated control flow automata (PCFA)*, we start with a discussion of PCFAs and then describe our CEGAR-based verification approach.

5.1 Predicated Control-Flow Automata

We represent each program using a generalized form of *control flow automaton (CFA)* that is commonly used in software model checking [Heizmann et al. 2013; Henzinger et al. 2004a, 2002]. A CFA is a directed graph where nodes correspond to program locations, and an edge from n to n' labeled with s indicates that the program transitions from location n to n' upon the execution of statement s. Predicated control flow automata (PCFA) augment CFA nodes with logical predicates:

Definition 5.1. **(PCFA)** A predicated control-flow automaton $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, S, \delta)$ where:

- Σ is the set of atomic program statements.
- S is a set of states, where each $s \in S$ is a pair $s = (l_m, \varphi)$. Here, l_m is a program location within method m, and φ is a formula over some first-order theory.
- δ is the transition relation $\delta \subseteq S \times \Sigma \times S$.

Notation. Given a state $s = (l, \varphi)$, we use $Loc(s)$ and $Pred(s)$ to denote l and φ respectively. $\text{Trans}(\mathcal{A})$ denotes the transition relation of $\mathcal{A}$. We use the notation $S{\downarrow}l = \{s \in S \mid Loc(s) = l\}$ to represent the subset of states in S that involve program location l. In addition, we write $In(l, \delta)$ (resp. $Out(l, \delta)$) to denote the in-coming (resp. out-going) edges of location of l. Finally, we say that state s' is reachable from state s, denoted as $\mathcal{A} \vdash s \rightsquigarrow s'$, if and only if $(s, \_, s') \in \delta$. As standard, we use $\mathcal{A} \vdash s \rightsquigarrow^* s'$ to represent the transitive closure of relation $\rightsquigarrow$.

5.2 Main Algorithm

Figure 11 presents our top-level verification algorithm. This procedure takes as input an (instrumented) program $\mathcal{P}$, represented as a mapping from methods to their PCFAs, as well as a context-free API protocol $\mathcal{G}_S$. The algorithm either returns "Verified" or a counterexample indicating an API misuse. As a convention, procedure names in small caps are formally defined later in this paper, whereas those in camel case are oracles that provide functionality that is orthogonal to our approach.

The main verification algorithm is a CEGAR loop that consists of the following steps. First, it calls procedure ConstructCFG (line 6) to obtain a context-free grammar $\mathcal{G}_\mathcal{P}$ that abstracts the relevant API usage of $\mathcal{P}$. Next, it checks whether there exists a word w that belongs in $\mathcal{L}(\mathcal{G}_\mathcal{P})$ but not in $\mathcal{L}(\mathcal{G}_S)$ (line 7). If this is not the case, the program must satisfy $\mathcal{G}_S$, so the algorithm returns "Verified" (line 14).

On the other hand, if there exists a word $w \in \mathcal{L}(\mathcal{G}_\mathcal{P}) \backslash \mathcal{L}(\mathcal{G}_S)$, we need to check whether w corresponds to a feasible execution path of $\mathcal{P}$. Given a derivation d of w, we convert this derivation to an execution path using an oracle called *derivation2path* (line 8). Here, we represent an execution path as a *nested trace* [Heizmann et al. 2010], which is a tuple $(\pi = \sigma_0 \ldots \sigma_n, \rightsquigarrow)$ where π is a sequence of program statements and $\rightsquigarrow$ is a so-called "nesting relation" between indices of π that associates matching call and return statements. That is, if $i \rightsquigarrow j$, then σ_j is a return statement and σ_i is its matching call statement. Given such a nested trace, we can easily check whether π is feasible by encoding it as an SMT formula and querying its satisfiability (line 9). If the path is feasible, then the algorithm returns π as a witness of API misuse.

In case π is infeasible, then word w is a spurious counterexample, and our algorithm refines the PCFA abstraction (lines 11-13) to eliminate the same spurious counterexample in the next iteration. To this end, we first make use of another oracle, *Interpolant*, which takes as input a nested word $(\pi = \sigma_0 \ldots \sigma_n, \rightsquigarrow)$ and returns an *inductive sequence of nested interpolants* $\mathcal{I} = [I_0, \ldots, I_{n+1}]$. Following Heizmann et al. [2010], we define nested interpolants as a sequence of predicates with

1: **procedure** VERIFY($\mathcal{P}$, $\mathcal{G}_S$)
2: **input:** $\mathcal{P} : M \rightarrow PCFA$, program.
3: **input:** $\mathcal{G}_S$, API-Protocol's context-free grammar.
4: **output:** *Verified* or Counterexample.
5: **while** *true* **do**
6: $\mathcal{G}_{\mathcal{P}} \leftarrow$ CONSTRUCTCFG($\mathcal{P}$)
7: **if** $\exists d.\ d \in InclusionCheck(\mathcal{G}_{\mathcal{P}}, \mathcal{G}_S)$ **then**
8: $(\pi, \rightsquigarrow) \leftarrow derivation2path(d)$
9: **if** $feasible((\pi, \rightsquigarrow))$ **then return** π
10: **else**
11: $\mathcal{I} \leftarrow Interpolant((\pi, \rightsquigarrow))$
12: $\Psi \leftarrow \left\{ l_m \mapsto \left\{ I_j \mid \begin{array}{c} I_j \in \mathcal{I}, \sigma_j \in \pi \\ Loc(\sigma_j) = l_m \end{array} \right\} \right\}$
13: $\mathcal{P} \leftarrow$ REFINE($\mathcal{P}, \Psi$)
14: **else return** *Verified*

Fig. 11. Verification Algorithm

the following properties: (1) $I_0 = true$, $I_{n+1} = false$. (2) If σ_i is not a return statement, then $sp(\sigma_i, I_i) \Rightarrow I_{i+1}$. (3) If σ_i is a return statement, then $sp(\sigma_i, I_i \wedge I_j) \Rightarrow I_{i+1}$ and $j \rightsquigarrow i$. Intuitively, the first property ensures that I can be used to prove infeasibility of $(\pi, \rightsquigarrow)$, whereas the latter two properties ensure that I is inductive.

After calculating a nested interpolant, the algorithm builds a mapping Ψ that groups interpolants by program location (line 12). That is, Ψ maps each program location to a set of predicates that should be tracked at that location. The REFINE procedure uses Ψ to determine how to clone program locations in the PCFAs such that $(\pi, \rightsquigarrow)$ is no longer feasible in the refined program abstraction.

We now state the following two theorems concerning the soundness and progress of our approach:

THEOREM 5.2. **(Soundness)** *Let $\mathcal{P}, \mathcal{P}'$ be the programs before and after the call to* REFINE *at line 13 respectively. Then, for every feasible execution path π in $\mathcal{P}$, there exists a derivation $d \in$ CONSTRUCTCFG($\mathcal{P}'$) such that $(\pi, \rightsquigarrow) = derivation2path(d)$.*

PROOF. The proofs of all theorems can be found in the extended version of the paper [Ferles et al. 2020]. □

THEOREM 5.3. **(Progress)** *Let t be a spurious counterexample returned by derivation2path and let $\mathcal{P}'$ be the resulting program after calling* REFINE *on program $\mathcal{P}$. Then, there does not exist a derivation $d \in$ CONSTRUCTCFG($\mathcal{P}'$) such that $t = derivation2path(d)$.*

PROOF. The proofs of all theorems can be found in the extended version of the paper [Ferles et al. 2020]. □

In the following subsections, we describe the REFINE (Section 5.3) and CONSTRUCTCFG (Section 5.4) procedures in more detail.

5.3 PCFA Refinement

Our PCFA refinement algorithm is summarized in Figure 12. Given program $\mathcal{P}$ and mapping Ψ from locations to predicates, the idea is to "clone" any program location $l \in \text{dom}(\Psi)$ based on the

```
1:  procedure Refine(𝒫, Ψ)
2:      input: 𝒫 : M → PCFA, program.
3:      input: Ψ : Loc → {Pred}, new predicates to track.
4:      output: Refined program with respect to Ψ
5:      for (l_m, Preds) ∈ Ψ do
6:          (Σ, S, δ) ← 𝒫[m]
7:          Φ ← CompleteCubes(Preds)
8:          S′ ← CloneStates(S, l_m, Φ)
9:          δ′ ← UpdateTransitions(δ, l_m, S′↓l_m)
10:         𝒫[m] ← (Σ, S′, δ′)
11:     return 𝒫
```

Fig. 12. Program Refinement Algorithm.

predicates $\Psi(l)$. Intuitively, the demand-driven cloning of program locations allows our method to be selectively path-sensitive and removes infeasible program paths encountered in previous iterations. Furthermore, our refinement algorithm is modular in the sense that we can refine the PCFA of each method independently.

In more detail, the Refine procedure iterates over each program location $l \in \text{dom}(\Psi)$ and determines which new states to create in the PCFA. Specifically, if $\Psi(l)$ contains n new predicates, then, for each state (l, ϕ) in the PCFA, we need to create 2^n new states, where each clone represents a copy of l under a different boolean assignment to the predicates in $\Psi(l)$. Towards this goal, the Refine procedure first invokes CompleteCubes (line 7) to generate a different boolean assignment as follows:

$$\text{CompleteCubes}(P) = \Big\{ \bigwedge_{i=1}^{|P|} c_i \mid c_i \in \{p_i, \neg p_i\}, p_i \in P \Big\}$$

In other words, CompleteCubes(P) yields a set Φ of (conjunctive) formulas such that every $\phi \in \Phi$ corresponds to a different boolean assignment to the predicates in P.

Next, given the new set of predicates Φ to track at location l, the procedure CloneStates (line 8) generates $|\Phi|$ clones of each state $(l, \phi) \in S$ as follows:

$$\text{CloneStates}(S, l, \Phi) = (S \setminus S{\downarrow}l) \ \cup \ \{(l, \varphi \wedge \varphi') \mid (l, \varphi) \in S, \ \varphi' \in \Phi\}$$

In other words, CloneStates removes all existing states (l, ϕ) associated with location l and then adds a new state $(l, \phi \wedge \phi')$ for each $\phi' \in \Phi$. Thus, if the PCFA contains n states for location l before refinement, then the refined PCFA contains $n \times |\Phi|$ states for location l.

Example 5.4. Consider the initial PCFA for method `acquire` from Fig. 3b and suppose $\Psi(a_3) = P = \{l1_{acq} = \$1\}$. In this case, we have $\Phi = \text{CompleteCubes}(P) = \{l1_{acq} = \$1, l1_{acq} \neq \$1\}$. Thus, CloneStates removes the original state (a_3, true) and generates two new states $(a_3, l1_{acq} \neq \$1)$ and $(a_3', l1_{acq} = \$1)$ as shown in Figure 6.

After creating the new states S', the Refine procedure updates the transition relation of the PCFA by invoking the UpdateTransitions function (line 9), defined as follows:

1: **procedure** ConstructCFG($\mathcal{P}$)
2: **input:** $\mathcal{P} : M \rightarrow PCFA$, program.
3: **output:** $\mathcal{G}_{\mathcal{P}}$, context-free grammar that abstracts $\mathcal{P}$.
4: $(T, N, R) \leftarrow (\emptyset, \emptyset, \emptyset)$
5: $\Theta \leftarrow \{(s, m) \mid s \in Exit(\mathcal{P}[m])\}$
6: **for** $(s_i, m) \in \Theta$ **do**
7: $(T_i, N_i, R_i, S_i) \leftarrow$ GenGrammar($\mathcal{P}[m], s_i, \Theta$)
8: $T \leftarrow T \cup T_i,\ N \leftarrow N \cup N_i,\ R \leftarrow R \cup R_i$
9: **if** IsMain(m) **then** $S \leftarrow S_i$
10: **return** (T, N, R, S)

Fig. 13. Context-Free Grammar Construction

$$\begin{aligned}\text{UpdateTransitions}(\delta, l, S') = \delta \setminus (&In(\delta, l) \cup Out(\delta, l)) \cup \\ &\{e = (s, \sigma, s') \mid s' \in S', (s, \sigma, \_) \in In(\delta, l),\ feasible(e)\} \cup \\ &\{e = (s', \sigma, s) \mid s' \in S', (\_, \sigma, s) \in Out(\delta, l),\ feasible(e)\}\end{aligned}$$

where $feasible((s_1, \sigma, s_2))$ is defined as $SAT(sp(\sigma, Pred(s_1)) \wedge Pred(s_2))$. In other words, UpdateTransitions first removes from δ all transitions involving location l. Then, for each new state $s' \in S'$ and for each incoming edge $(s, \sigma, \_)$ to location l, it adds a new edge (s, σ, s') as long as the annotation of the new state s' is consistent with the annotation of the source node, $Pred(s)$, and the semantics of statement σ. Outgoing edges from location l are also updated analogously.

Example 5.5. Consider again the new states at the end of method `acquire`. Observe that UpdateTransitions will not add an edge between states a_1,a_3' and a_2,a_3 in the refined version of the PCFA (shown in Fig. 6b) because *feasible* returns false for these edges.

5.4 Context-Free Grammar Construction

In this section, we describe how to extract a context-free grammar from the PCFAs. As explained earlier, the main idea is to represent relevant API invocations as terminals in the grammar so that words generated by the CFG correspond to all possible sequences of API calls issued by the program. Towards this goal, we introduce one non-terminal symbol for each PCFA state and generate CFG productions according to the PCFA transitions. The resulting CFG abstraction is (selectively) path-sensitive in that we introduce as many non-terminal symbols for a method as it has exit states. Intuitively, different non-terminals for method m correspond to different "summaries" conditioned upon facts that hold at m's call sites.

The ConstructCFG procedure is described in more detail in Figure 13. It generates the program's CFG abstraction by iterating over every exit state s of each method m and constructs a separate grammar for (s, m) using the call to GenGrammar at line 7. The CFG for the whole program is obtained as the union of all of the individual grammars, and the start symbol for $\mathcal{G}_{\mathcal{P}}$ is the one associated with `main`.

Figure 14 summarizes the GenGrammar procedure using inference rules of the following shape:

$$\mathcal{A}, c, \Theta \vdash \Delta_1, \ldots, \Delta_n$$

$$(1)\quad \frac{(s,\sigma,s') \in \mathsf{Trans}(\mathcal{A}) \quad \neg \mathit{callStmt}(\sigma) \quad \mathcal{A} \vdash s' \rightsquigarrow^* c \quad \mathit{Pred}(c) = \varphi}{\mathcal{A}, c, \Theta \vdash \{\mathcal{S}_\varphi, \mathcal{S}'_\varphi\} \subseteq N_c \quad \mathcal{S}_\varphi \to \mathcal{S}'_\varphi \in R_c}$$

$$(2)\quad \frac{(s,\sigma,s') \in \mathsf{Trans}(\mathcal{A}) \quad \sigma = \mathit{api\_call}\ m(\vec{v}) \quad \mathcal{A} \vdash s' \rightsquigarrow^* c \quad \mathit{Pred}(c) = \varphi}{\mathcal{A}, c, \Theta \vdash \{\mathcal{S}_\varphi, \mathcal{S}'_\varphi\} \subseteq N_c \quad \sigma \in T_c \quad \mathcal{S}_\varphi \to \sigma\, \mathcal{S}'_\varphi \in R_c}$$

$$(3)\quad \frac{\begin{array}{c}\mathit{feasible}(e,\varphi') \quad \mathit{Pred}(c) = \varphi \\ e = (s,\sigma,s') \in \mathsf{Trans}(\mathcal{A}) \quad \sigma = \mathit{call}\ m'(\vec{v}) \quad (c', m') \in \Theta \quad \mathcal{A} \vdash s' \rightsquigarrow^* c \quad \varphi' = \mathit{Pred}(c')\end{array}}{\mathcal{A}, c, \Theta \vdash \{\mathcal{S}_\varphi, \mathcal{S}'_\varphi\} \subseteq N_c \quad \mathcal{S}_\varphi \to \mathcal{M}'_{\varphi'}\, \mathcal{S}'_\varphi \in R_c}$$

$$(4)\quad \frac{s \in \mathit{Entry}(\mathcal{A}) \quad \mathcal{A} \vdash s \rightsquigarrow^* c \quad \varphi = \mathit{Pred}(c)}{\mathcal{A}, c, \Theta \vdash \mathcal{M}_\varphi \to \mathcal{S}_\varphi \in R_c \quad \mathcal{M}_\varphi \in N_c \quad S_c = \mathcal{M}_\varphi} \qquad (5)\quad \frac{\varphi = \mathit{Pred}(c)}{\mathcal{A}, c, \Theta \vdash C_\varphi \to \epsilon \in R_c}$$

Fig. 14. Rules for constructing CFG = (T_c, N_c, R_c, S_c) given a exit state c in PCFA $\mathcal{A} = (\Sigma, S, \delta)$, and set Θ. For a PCFA state s with predicate φ, the symbol $\mathcal{S}_\varphi$ denotes the corresponding non-terminal in the grammar.

Here, the left-hand side of the turnstile represents the arguments of the GenGrammar procedure, and each Δ_i is a set inclusion constraint for the CFG symbols and productions. In more detail, $\mathcal{A}$ is the PCFA for the current method, c is an exit state in $\mathcal{A}$, and Θ is a set of pairs (s, m) where s is an exit state in method m's PCFA. (As we will see shortly, GenGrammar uses Θ to generate grammar productions for method calls.) Given a state s in the PCFA and predicate φ labeling exit state c, GenGrammar generates a non-terminal $\mathcal{S}_\varphi$ for each state in the PCFA.

Statements. The first rule in Figure 14 applies to all statements that are *not* function calls. Since atomic statements other than API calls are not relevant to our abstraction, this rule only captures control-flow dependencies. Specifically, let (s, σ, s') be a PCFA edge where σ is a non-call statement. First, we introduce non-terminals $\mathcal{S}_\varphi, \mathcal{S}'_\varphi$ for states s, s' and add a production $\mathcal{S}_\varphi \to \mathcal{S}'_\varphi$ to capture that s' is a successor of s. Observe that this rule (as well as the next two rules) have $\mathcal{A} \vdash s' \rightsquigarrow^* c$ as a premise because non-terminals $\mathcal{S}_\varphi, \mathcal{S}'_\varphi$ should only be added to the grammar if s, s' are backward-reachable from exit state c.

Example 5.6. The production $\mathcal{A}_{1,\phi_1} \to \mathcal{A}_{2,\phi_1}$ in Fig. 7 is generated using the *Stmt* rule based on the PCFA from Fig 6.

API. The next rule generates productions for API calls. This rule is similar to the previous one but with two key differences: First, it also adds σ to terminals T_c. Second, it generates the production $\mathcal{S}_\varphi \to \sigma\mathcal{S}'_\varphi$ instead of $\mathcal{S}_\varphi \to \mathcal{S}'_\varphi$ because σ is relevant to the program's API usage.

Example 5.7. Consider the production $\mathcal{A}_{2,\phi_1} \to \texttt{\$1.lock()}\ \mathcal{A}'_{3,\phi_1}$ from Figure 7. This production is generated due to the PCFA transition $(a_2, \texttt{\$1.lock()}, a'_3)$ from Figure 6.

Call. The third rule applies to PCFA edges (s, σ, s') where σ is a call to method m'. Since there are multiple "clones" of m', let us consider one specific clone c' with "summary" φ'. In this case, we generate the production $\mathcal{S}_\varphi \to \mathcal{M}'_{\varphi'}\mathcal{S}'_\varphi$, where $\mathcal{M}'_{\varphi'}$ is the start symbol for the grammar associated with this clone of m'. However, since predicate φ' may be inconsistent with PCFA transition (s, σ, s'), we first check whether this *particular* clone of m' is feasible at this call site. This is done by requiring $\mathit{feasible}(e, \varphi')$, defined as follows:

$$\mathit{feasible}((s, \texttt{call m'}(\vec{v}), s'), \varphi') \equiv \mathit{SAT}(\mathit{Pred}(s) \wedge \mathit{Pred}(s') \wedge \varphi')$$

Example 5.8. Consider the PCFAs from Figure 6. Here, the production $\mathcal{F}_3 \rightarrow Acquire_{\{l1_{acq}=\$1\}}\mathcal{F}_4$ belongs to $\mathcal{G}_\mathcal{P}$ because we have $feasible(e, l1_{acq} = \$1)$ for the PCFA edge e from f_3 to f_4. On the other hand, there is no production $\mathcal{F}_3 \rightarrow Acquire_{\{l1_{acq}=\$1\}}\mathcal{F}'_4$ because $l1_{acq} = l \wedge l \neq \$1 \wedge l1_{acq} = \$1$ is unsatisfiable.

Entry and exit. The last two rules in Figure 14 deal with the entry and exit states of the PCFA. Specifically, for any entry state s of the PCFA that is backward-reachable from the target exit state c, we add a production $\mathcal{M}_\varphi \rightarrow \mathcal{S}_\varphi$, where $\mathcal{M}_\varphi$ corresponds to the start symbol of the grammar. For exit state c, we just add the empty production $\mathcal{C}_\varphi \rightarrow \epsilon$.

Example 5.9. For the PCFA from Figure 6b, we add the production $Acquire_{\phi_1} \rightarrow \mathcal{A}_{0,\phi_1}$ because a_0 is an entry state that is backward-reachable from state a'_3. Similarly, we add a production $\mathcal{A}'_{3,\phi_1} \rightarrow \epsilon$ for exit state a'_3.

6 IMPLEMENTATION

We implemented our approach in a prototype called CFPChecker for analyzing Java programs. CFPChecker is implemented in Java on top of the Soot infrastructure [Vallée-Rai et al. 1999] and uses the technique of Madhavan et al. [2015] to perform grammar inclusion checks. Our implementation also makes use of SMTInterpol [Christ et al. 2012] to obtain nested interpolants and leverages Z3 [De Moura and Bjørner 2008] to determine satisfiability.

In the remainder of this section, we discuss some design choices and optimizations that were omitted from the technical presentation.

Slicing input programs. Before running the verification algorithm presented in Section 5, CFPChecker uses slicing to improve scalability. Specifically, we first identify all calls to the API whose usage is being checked and then compute a backward slice of the program with respect to those statements [Sridharan et al. 2007; Weiser 1981].

From words to execution paths. As mentioned in Section 5, we assume that the *InclusionCheck* method returns a derivation $d \in \mathcal{G}_\mathcal{P}$ for a word $w \in \mathcal{L}(\mathcal{G}_\mathcal{P}) \setminus \mathcal{L}(\mathcal{G}_S)$. In practice, $\mathcal{G}_\mathcal{P}$ tends to be highly ambiguous, so obtaining such a derivation for w can be computationally expensive. To address this issue, we first convert $\mathcal{G}_\mathcal{P}$ to Chomsky Normal Form (CNF) [Chomsky 1959] for which there is a polynomial algorithm for obtaining a derivation [Hopcroft 2008], and we then map this derivation back to the original grammar. While mapping the CNF derivation to the original grammar is not polynomial time, we have found this strategy to work much better in practice compared to directly searching for a derivation in the original grammar.

Handling pointers. In our implementation, we model the heap by using a fairly standard array-based encoding that has been popularized by ESC-Java [Flanagan et al. 2002]. Specifically, we introduce an array for each field and model loads and stores using select and update functions in the theory of arrays.

Obtaining PCFAs. Before generating the PCFA of a method, we first perform a program transformation similar to the one described by Ball et al. [2005] to enable polymorphic predicate abstraction. Specifically, for each method in the program, we generate auxiliary variables, referred to as *symbolic constants* in prior work, that track the initial value of variables on method entry. This transformation allows computing polymorphic interpolants that can be reused across call sites.

Optimizations. Rather than introducing one non-terminal symbol for every *program location*, we instead introduce one non-terminal for each *basic block* in order to make the resulting context-free grammar smaller. Also, since the refinement algorithm may issue an exponential number of

satisfiability queries, we issue SMT queries in parallel whenever possible and memoize the results of Z3 queries. Finally, since mapping parse trees from the CNF grammar back to the original version can be a performance bottleneck, we memoize partial results between refinement iterations.

Limitations. Similar to other verification tools, CFPChecker models several Java features (e.g., exceptions, reflection) in a "soundy" way [Livshits et al. 2015]. Furthermore, since CFPChecker models program semantics using the combined theory of arrays and linear integer arithmetic, it also conservatively over-approximates operations that fall outside of this theory. In particular, CFPChecker introduces appropriate uninterpreted functions to model operations that involve non-integer variables (e.g., floats, doubles, etc.).

7 EVALUATION

To evaluate CFPChecker, we collected real-world use cases of Java APIs and conducted experiments designed to answer the following research questions:

RQ1: Can CFPChecker verify the correct usage of popular Java APIs in real-world clients?
RQ2: Does the proposed technique advance the state-of-the-art in software verification?

To answer these questions, we conduct two sets of experiments. For our first experiment, we collect five popular Java APIs with context-free specifications and evaluate CFPChecker on 10 widely-used Java programs that leverage at least one of these five APIs. In our second experiment, we compare CFPChecker against existing verification tools. However, since there is no off-the-shelf technique that can directly verify correct usage of context-free API protocols, we instrument (simplified versions of) these 10 Java programs with suitable assertions that enforce correct API usage, and we then try to discharge these assertions using state-of-the-art verification and model checking tools.

All of our experiments are run on an Intel Xeon CPU E5-2640 v3 @ 2.60GHz machine with 132 GB of memory running the Ubuntu 14.04.1 operating system.

7.1 API Specifications & Benchmarks

For our evaluation, we consider the following five popular Java APIs whose correct usage is defined by a context-free specification:

(1) **ReentrantLock**: a widely-used Java API that implements a reentrant lock
(2) **WakeLock**: a popular Android API that allows the client application to keep the Android device awake
(3) **WifiLock**: another Android API that allows the applications to keep the Wi-Fi radio awake
(4) **Canvas**: a graphics API (also for Android) that allows clients to create views and animations
(5) **JsonGenerator**: a serialization library that allows serializing Java objects as JSON documents

Specifications. Table 1 presents the context-free protocols that clients of these APIs must adhere to. As used as a running example throughout the paper, `ReentrantLock` requires calls to `acquire` and `release` to be balanced, and failure to follow this protocol results in deadlocks. The next two APIs, namely `WakeLock` and `WifiLock`, have the exact same specification and can be used in two different modes of operation, reference-counted and non-reference-counted. The specification for the first mode is the same as `ReentrantLock` (i.e., each call to `acquire` must be matched by a call to `release`). On the other hand, the second mode is enabled by the call `setRefCnt(false)` and requires the usage pattern to be of the form $\texttt{acquire}^n\ \texttt{release}^m$ where $m \leq n$ and $n \geq 1 \longrightarrow m \geq 1$. For both the `WakeLock` and `WifiLock` APIs, failure to follow the protocol causes resource leaks (e.g., the application drains the phone's battery). For the Android `Canvas` API, its documentation states "It is an error to call `restore()` more times than `save()` was called."; thus, its specification is of the form

Table 1. Java API Protocol Specifications

API Name	Specification
ReLock	$S \rightarrow$ `$1.acquire()` S `$1.release()` S $\mid \epsilon$
Wifi & Wake Lock	$S \rightarrow RC \mid$ `$1.setRefCnt(false)` NC $NC \rightarrow \epsilon \mid NA$ `$1.release()` $NA \rightarrow$ `$1.acquire()` NA `$1.release()` $NA \mid$ `$1.acquire()` NA $\mid$ `$1.acquire()` $RC \rightarrow$ `$1.acquire()` RC `$1.release()` $RC \mid \epsilon$
Canvas	$S \rightarrow \epsilon \mid$ `$1.save()` S `$1.restore()` S $\mid$ `$1.save()` S
Json Gen.	$S \rightarrow \epsilon \mid Obj \mid Arr \mid$ `$1.writeString()` $\mid$ `$1.writeNumber()` $\mid$ `$1.writeBoolean()` $Obj \rightarrow$ `$1.writeStartObject()` Fld `$1.writeEndObject()` $Fld \rightarrow$ `$1.writeFieldName()` $S\ Fld \mid \epsilon$ $Arr \rightarrow$ `$1.writeStartArray()` $Vals$ `$1.writeEndArray()` $Vals \rightarrow \epsilon \mid S\ Vals$

`save`^{n} `restore`^{m} where $m \leq n$. Failure to follow this protocol results in a run-time exception. The last API, called `JsonGenerator`, has a relatively complex specification and requires clients to call API methods (e.g., `writeStartObject()`, `writeEndObject()`, etc.) in accordance with the JSON schema, that is, calls that start (e.g., `writeStartObject()`) and end (e.g., `writeEndObject()`) a JSON element must be matched and properly nested. Failure to follow this protocol results in the generation of invalid JSON files.

Clients. To evaluate our approach on realistic usage scenarios of these libraries, we collected ten open-source Java programs that use these APIs. The clients used in our evaluation are widely-used programs such as Hadoop/MapReduce (a distributed computing framework), ExoPlayer (an Android media player), ConnectBot (secure shell client), Netflix Hystrix (a fault tolerance library for distributed environments), etc. These applications contain an average of 571 classes and 36,390 lines of Java code (equivalently, 56,114 Soot bytecode instructions). Recall that we first slice the input program before we run any of the verifiers (Section 6). The effectiveness of slicing varies across different benchmarks with the resulting slices containing between 3-126 classes.

7.2 Results for CFPChecker

Table 2 summarizes our main experimental results for CFPChecker. As we can see from the "Output" column, two of the benchmarks (namely, ExoPlayer and ConnectBot) actually misuse at least one API. For these benchmarks (indicated with the $\lightning$ symbol), we also construct a correct variant (indicated without the $\lightning$ symbol) by manually repairing the original bug. We now summarize the key take-away lessons from this evaluation.

Table 2. Results for CFPChecker. Under the “output" column, "Cex" denotes a counterexample and "Safe" indicates that the benchmark was verified. Total time indicates end-to-end running time in seconds, and “Incl. check” shows the time spent performing grammar inclusion checking queries. “# Steps”: number of refinement steps, “Preds / BB”: Average and max predicates tracked per basic block, “# Preds”: total number of predicates tracked.

Benchmark Info		CFPChecker Statistics					
	Benchmark	Output	Total Time	Incl. Check	# Steps	Preds / BB (Avg/Max)	# Preds
Wifi	ExoPlayer (↯)	Cex	63.8	0.3	27	2/6	40
	ExoPlayer	Safe	35.2	0.4	20	1/4	44
WakeLock	ExoPlayer (↯)	Cex	66.6	0.3	24	2/6	40
	ExoPlayer	Safe	47.0	0.5	20	2/6	39
	ConnectBot (↯)	Cex	392.0	9.3	42	3/9	107
	ConnectBot	Safe	2336.5	18.3	48	3/12	133
ReLock	Hystrix	Safe	20.7	0.3	9	1/3	21
	Guice	Safe	221.5	2.4	25	3/9	92
	Bitcoinj	Safe	3175.3	28.0	79	1/5	115
Canvas	Glide	Safe	562.6	544.7	8	1/3	19
	RxTool	Safe	56.4	43.8	1	1/1	3
	Litho	Safe	14.4	0.5	5	1/3	11
Json	Hadoop	Safe	140.2	64.5	49	1/4	65
	Hystrix-1	Safe	64.4	2.7	48	2/4	62
	Hystrix-2	Safe	24.2	0.6	31	1/4	55

Verification results for correct benchmarks. CFPChecker is able to successfully verify all benchmarks that correctly use the relevant API. On average, CFPChecker takes 9.3 minutes to verify each application, and its median verification time is 60.4 seconds. Most of the benchmarks require a significant number of refinement steps, with 22.5 being the median number of iterations.

Counterexamples for buggy benchmarks. As shown in Table 2, CFPChecker reports three API protocol violations. Two of these violations are in ExoPlayer, which misuses both the WifiLock and WakeLock libraries, and the other violation is in ConnectBot, which misuses WakeLock. Using the counterexamples reported by CFPChecker, we were able to identify the root causes of these errors. Interestingly, all three violations share the same root cause. In particular, ExoPlayer and ConnectBot both call the `acquire` method in `onStart` and the corresponding `release` method in `onStop` of an Android Activity [Documentation 2020]; however, they fail to release the lock in the `onPause` method. Since the Android framework may kill a paused activity when there is memory pressure (see Figure 15), the calls to `acquire` and `release` are not guaranteed to be matched. Thus, this bug can result in resource leaks in the form of unintended battery usage. One simple way to fix this issue is to move the `acquire` and `release` calls to the `onResume` and `onPause` methods instead. In fact, a later version of the ConnectBot application fixes the bug in exactly this way; however, CFPChecker identified a previously unknown issue in ExoPlayer.

Table 3. Results for other safety-checking tools on simplified benchmarks using a time limit of 8 hours and memory limit of 16 GB per benchmark. Values in the "Out." columns have the following meaning: Cex: feasible counterexample found, Safe: no violations found, Unknown: unable to produce neither a counterexample nor a proof of correctness, TO: timeout, OM: out of memory. We use a "-" to indicate that a value is not applicable. All execution times are in seconds.

		JayHorn			JPF-BugFinder			JPF-Verifier		
	Bench.	Out.	Correct Output?	Time	Out.	Correct Output?	Time	Out.	Correct Output?	Time
Wifi	ExoPlayer (↯)	Unknown	✗	1212.4	Safe	✗	0.6	TO	✗	-
	ExoPlayer	Unknown	✗	1119.0	-	-	-	TO	✗	-
WakeLock	ExoPlayer (↯)	Unknown	✗	1105.3	Safe	✗	0.5	TO	✗	-
	ExoPlayer	Unknown	✗	1162.8	-	-	-	TO	✗	-
	ConnectBot (↯)	Unknown	✗	333.0	Safe	✗	0.7	TO	✗	-
	ConnectBot	Unknown	✗	164.0	-	-	-	TO	✗	-
ReLock	Hystrix	Unknown	✗	12530.5				TO	✗	-
	Guice	Safe	✓	2383.1	-	-	-	TO	✗	-
	Bitcoinj	OM	✗	-	-	-	-	TO	✗	-
Canvas	Glide	TO	✗	-	-	-	-	TO	✗	-
	RxTool	TO	✗	-	-	-	-	TO	✗	-
	Litho	OM	✗	-	-	-	-	TO	✗	-
Json	Hadoop	OM	✗	-	-	-	-	TO	✗	-
	Hystrix-1	Safe	✓	931.7	-	-	-	TO	✗	-
	Hystrix-2	OM	✗	-	-	-	-	TO	✗	-

Summary. As these experiments indicate, verifying the correct usage of context-free API protocols is of practical relevance in real-world applications. Our results demonstrate that CFPChecker is practical enough to verify the correct usage of context-free API protocols in widely-used Java applications and that it can provide useful counterexamples when the property is violated.

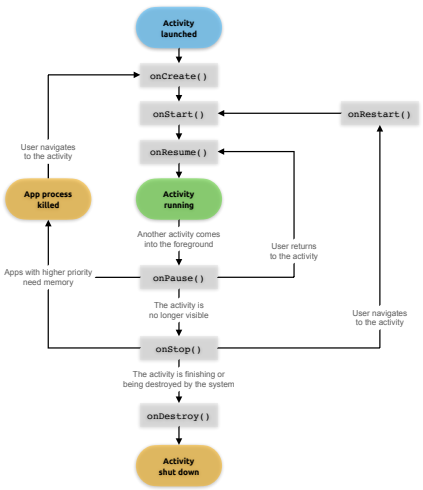


Fig. 15. Android lifecycle callbacks

7.3 Comparison with Baselines

Since there is no existing tool for verifying correct usage of context-free protocols, we cannot *directly* compare our approach against existing baselines. Thus, we construct our own baselines using the following strategy: First, we instrument each program with suitable assertions that enforce correct API usage (as explained below). Then, we try to discharge these assertions using existing safety verifiers. In this section, we report on our experience implementing and evaluating these baselines using JayHorn [Kahsai et al. 2016] and JavaPathFinder [Artho and Visser 2019] as the assertion checking back-ends. Note that JayHorn is a state-of-the-art Java

verification tool based on constrained Horn clause solvers, and JavaPathFinder is a mature model checking tool for Java developed by NASA.

Assertion instrumentation. As mentioned earlier, the goal of our instrumentation is to generate a program P' such that P' is free of assertion failures if and only if the original program P obeys a given context-free API protocol. One obvious way to perform this instrumentation is to represent the API protocol using a push-down automaton (PDA) and then introduce variables that keep track of the PDA's state and stack contents. In fact, this strategy has been used in prior work for performing run-time checking of correct API usage [Chen and Roşu 2007; Jin et al. 2012; Meredith et al. 2010]. However, since static techniques are typically not very good at reasoning about dynamically allocated data structures (e.g., arrays), we instead manually perform *API-specific instrumentation* that avoids introducing arrays whenever possible. For example, for `ReentrantLock`, we only introduce an integer counter `c` that is incremented (resp. decremented) on calls to `lock` (resp. `unlock`). Then, to enforce the protocol, we assert that `c` is positive when `unlock` is called and that it is zero at the end. Using similar strategies, we can perform instrumentation using *only integer variables* for all APIs except one (`JsonGenerator`).

Please note that the instrumentation strategy described above requires human ingenuity and cannot be used to *automatically* check arbitrary API protocols. However, it is designed to be *as favorable as possible* to assertion checking tools and represents the best possible scenario for existing verifiers.

Slicing and pre-processing. Recall from Section 6 that CFPChecker incorporates a slicing step to enable better scalability. To ensure a fair comparison, we use the exact same slicing procedure before feeding the instrumented programs to the assertion checking tools. However, even the slices contain several features that cannot be handled by at least one of these two tools. For instance, if we provide the generated slices to JayHorn as-is, it crashes on most benchmarks. Similarly, JavaPathFinder throws an exception whenever it encounters a call to a method whose source code is not available. Therefore, in order to use JayHorn and JavaPathFinder as our assertion-checking back-ends, we further manually simplified our benchmarks from Section 7.1 in a way that preserves the relevant API usage-related behavior.

Configurations of JavaPathFinder. The JPF tool can be configured in several different ways. In this experiment, we use two configurations of JPF for the assertion-checking back-end. The first variant, henceforth called JPF-BugFinder, is a version of Java Pathfinder that is configured with the default settings for SVCOMP [Artho and Visser 2019]. Note that these settings are suitable for bug finding but not for verification. To use JavaPathFinder as a verifier, we also consider a second variant where we do not restrict its search space. We refer to this variant as JPF-Verifier. Since JPF-BugFinder is *not* a verifier, we only evaluate it on the buggy benchmarks.

Overall results. The results of our comparison against these three baselines (JayHorn, JPF-BugFinder, and JPF-Verifier) are presented in Table 3. The key take-away from this experiment is that none of the three baselines are effective at successfully verifying (or finding bugs in) our experimental benchmarks despite manual simplification and instrumentation. In what follows, we describe the results for each of the three baselines in more detail.

Results for JayHorn. JayHorn verifies only 2 of the 15 benchmarks. For 7 benchmarks, JayHorn reports a possible assertion violation, but is unable to provide a counterexample. For the remaining 6 benchmarks, JayHorn either fails to terminate within the 8-hour time limit or runs out of memory. Surprisingly, one of the benchmarks (Hystrix-1) that can be verified by JayHorn uses the complex

`JsonGenerator` API. We conjecture that JayHorn can verify this benchmark more easily because it does not involve recursion and all relevant API usage is confined within a single method.

Results for JPF. When using JPF as a bug finder with the default SV-COMP settings, it fails to find the assertion violations in the three buggy benchmarks and reports them as safe. This result suggests that the API protocol violations in the buggy benchmarks are non-trivial to find. On the other hand, JPF-Verifier fails to terminate within the eight hour time limit on any benchmark, and it also fails to find the errors in the three buggy benchmarks.

8 RELATED WORK

We now survey prior work related to this paper and highlight their differences from our approach.

Typestate analysis. Most prior work on checking correct API usage focuses on protocols that can be expressed as a regular language [Ball et al. 2006; Fink et al. 2008; Strom and Yemini 1986]. This problem is commonly known as *typestate analysis* [Strom and Yemini 1986], and researchers have proposed many different approaches to solve this problem ranging from language-based solutions [Aldrich et al. 2009; Bierhoff and Aldrich 2007; DeLine and Fähndrich 2004; Garcia et al. 2014] to program analysis [Bodden 2010; Bodden and Hendren 2012; Fink et al. 2008] and model checking [Ball et al. 2006; Ball and Rajamani 2001] to bug finding [Joshi and Sen 2008; Yu et al. 2018] and run-time verification [Allan et al. 2005; Chen and Roşu 2007]. Some prior works have also proposed various generalizations of typestate properties, such as multi-object protocols [Beckman et al. 2011; Pradel et al. 2012b].

Run-time checking for context-free properties. There have been some proposals, particularly in the context of run time techniques, for checking correct usage of APIs with context-free specifications. In particular, these techniques [d'Amorim and Havelund 2005; Jin et al. 2012; Martin et al. 2005; Meredith et al. 2010] instrument the program with monitors that keep track of PDA states and dynamically check for property violations. As shown in our experiments, such an instrumentation-based approach does not work well for static verification.

Interface grammars. Prior work has proposed *interface grammars* for specifying the sequences of method invocations that are allowed by a library [Hughes and Bultan 2008]. Given an interface grammar for a component, this technique generates a stub that can be used to analyze clients of that component. While this work addresses a somewhat different problem, their technique bears similarities to our instrumentation-based baseline, which, as shown in our evaluation, does not work well in our setting.

CEGAR. Similar to all CEGAR approaches [Clarke et al. 2000, 2003; Grebenshchikov et al. 2012; Gurfinkel et al. 2015; Heizmann et al. 2018; Henzinger et al. 2004b, 2002, 2003], our method starts with a coarse abstraction and iteratively refines it based on spurious counterexamples. However, our method differs from most CEGAR-based techniques in that we abstract the program using a context-free grammar and perform refinement by adding new non-terminals and productions to the grammar.

Abstracting programs with CFGs. Similar to our approach, prior work on has explored abstracting programs using context-free grammars. For example, Long et al [Long et al. 2012] use CFG inclusion checking to prove assertions in concurrent programs; however, their approach does *not* refine the program's CFG abstraction. Instead, they use a CEGAR approach to solve the CFG inclusion checking problem through a sequence of increasingly more precise regular approximations. Furthermore, since they address a different problem, their CFG abstraction is quite different from ours. Another

related approach in this space is the work by Ganty et al. [2010] which also abstracts recursive multi-threaded programs with a context-free grammar. In contrast to our work, they *under-approximate* the reachable state space of recursive multi-threaded programs by generating a succession of bounded languages that under-approximate the program's CFG.

Interpolants. Similar to many CEGAR-based techniques [Gurfinkel et al. 2015; Henzinger et al. 2004b; McMillan 2005, 2006], our method also uses *Craig interpolation* to learn new predicates when a spurious counterexample is discovered. Given an unsatisfiable formula $\phi \wedge \psi$, a Craig interpolant is another formula χ such that $\phi \Rightarrow \chi$ is valid and $\psi \wedge \chi$ is unsatisfiable. Prior work has proposed many variants of Craig interpolation, including sequence interpolants [Henzinger et al. 2004b], tree interpolants [Blanc et al. 2013], nested interpolants [Heizmann et al. 2010], and DAG interpolants [Albarghouthi et al. 2013]. In this paper, we leverage the notion of nested interpolants introduced in Heizmann et al. [2010] to infer useful predicates for recursive procedures; however, our refinement procedure uses these nested interpolants in a very different way.

Control flow refinement. Our refinement technique bears similarities to prior work on control-flow refinement [Balakrishnan et al. 2009; Cyphert et al. 2019; Flores-Montoya and Hähnle 2014; Gulwani et al. 2009]. Similar to CFPChecker, these techniques clone program locations in order to exclude infeasible paths from their program abstraction. However, all of these techniques abstract the program using a regular language, and, with the exception of Flores-Montoya and Hähnle [2014], they apply control-flow refinement within a single procedure and only inside loops. On the other hand, Flores-Montoya and Hähnle [2014] refines cost equations rather than the program abstraction. In contrast to all of these techniques, our technique refines the CFG abstraction, performs cloning inter-procedurally, and supports arbitrary recursion.

Directed proof generation. Directed proof generation (DPG) techniques simultaneously maintain an under- and an over-approximation of the program and evolve them in a synergistic way [Thakur et al. 2010]. Specifically, the under-approximation is used to find feasible counterexamples and learn new predicates which refine the over-approximation. Conversely, the over-approximation is used to generate proofs and guides counterexample search to paths that are more likely to fail. Similar to our technique, DPG-like approaches [Beckman et al. 2010; Godefroid et al. 2010; Gulavani et al. 2006; Thakur et al. 2010] annotate their control-flow representation with logical predicates and clone program locations. Our approach differs from these techniques in the way it discovers potential counterexamples and new predicates. In particular, CFPChecker performs an inclusion check between two context-free languages in order to discover a potential API violation and uses interpolation to discover new predicates. In contrast, DPG techniques use a combination of graph reachability and test-case generation.

Equivalence of context-free languages. Our approach leverages prior work on checking containment between two context-free languages [Harrison et al. 1979; Korenjak and Hopcroft 1966; Madhavan et al. 2015; Olshansky and Pnueli 1977]. While checking inclusion between arbitrary context-free languages is known to be undecidable, prior work has studied decidable fragments, such as LL(k) grammars [Olshansky and Pnueli 1977]. Our implementation makes use of the algorithm by Madhavan et al. [2015], which in turn extends prior algorithms for LL grammars. While our technique is orthogonal to checking context-free language containment, it would directly benefit from advances and new algorithms that address this problem.

CFL reachability. CFL reachability techniques represent inter-procedural control flow using a graph representation and then filter out paths that do not conform to valid call-return structures [Reps et al. 1995]. This formulation has been used to express several fundamental program

analyses, such as context-sensitive pointer analysis [Sridharan et al. 2005; Xu et al. 2009]. However, adding another level of sensitivity (e.g., field-sensitivity,) requires solving two separate CFL reachability problems on the same execution path, which is known to be undecidable [Reps 2000]; hence many techniques over-approximate one of the two CFL reachability problems [Chatterjee et al. 2017; Li et al. 2020; Späth et al. 2019; Sridharan and Bodík 2006; Sridharan et al. 2005; Xu et al. 2009; Zhang et al. 2013] or propose a more precise generalization of CFL reachability [Tang et al. 2015; Zhang and Su 2017]. Similar to these techniques, we also need to reason about two context-free properties, namely matching call-return statements and matching between calls to API methods. However, this work addresses a somewhat different problem: instead of filtering out execution paths that do not belong to both context-free languages, our technique verifies that *every* API sequence generated by an execution path with a valid call-return structure belongs to the context-free specification.

Visibly pushdown automata. Many model checking techniques use variants of pushdown automata, such as *visibly pushdown automata* (VPAs) or *nested word automata* (which are equally expressive), to reason about inter-procedural control flow [Alur and Madhusudan 2004; Chen and Wagner 2002; Esparza et al. 2003; Henzinger et al. 2002]. Visibly pushdown and nested word automata are less expressive compared to PDAs; however, they enjoy various decidability and closure properties for operations like intersection and complement. However, VPAs cannot capture two separate context-free properties on the same execution path, which is required by our technique.

There have been some theoretical studies that extend VPAs to use multiple stacks [Carotenuto et al. 2007; Torre et al. 2007; Torre et al. 2013], and such multi-stack VPAs are significantly more expressive compared to standard VPAs. For example, 2-VPAs [Carotenuto et al. 2007] (i.e., VPAs with two stacks) accept some context-sensitive languages that are not context-free and some context-free languages that are not accepted by any VPA. We believe that it would be possible to solve the problem addressed in this paper using 2-VPAs, however, the emptiness problem for 2-VPAs is also undecidable.

9 CONCLUSION

We presented a technique for verifying the correct usage of context-free API protocols. Our approach abstracts the program as a context-free grammar representing feasible API call sequences and checks whether this CFG is contained inside the specification CFG. Our method follows the CEGAR paradigm and lazily refines the CFG by introducing new productions and non-terminals that represent clones of methods and program locations.

We implemented the proposed method in a tool called CFPChecker and performed an experimental evaluation on 10 widely-used Java applications that utilize at least one of 5 popular APIs with context-free specifications. Our evaluation shows that CFPChecker can verify all correct usage patterns while finding counterexamples for the buggy clients. We also implement and evaluate three baselines that reduce this problem to assertion checking and then use oft-the-shelf safety checking tools to discharge these assertions. Our experience with these baselines suggests that our method is more amenable to automation than alternative approaches that reduce the problem to assertion checking.

ACKNOWLEDGMENTS

We would like to thank our shepherd Pierre Ganty, the anonymous reviewers, Kenneth McMillan, Swarat Chaudhuri, and the members of the UToPiA group for their insightful feedback. This work is supported in part by NSF Award #1453386 and DARPA Award #FA8750-20-C-0208.

REFERENCES

Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. 2013. UFO: verification with interpolants and abstract interpretation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 637–640.

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 1015–1022.

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2005. Adding trace matching with free variables to AspectJ. In *OOPSLA*.

Rajeev Alur and Parthasarathy Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, 202–211.

Cyrille Artho and Willem Visser. 2019. Java Pathfinder at SV-COMP 2019 (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 224–228.

Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. 2015. Towards secure integration of cryptographic software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 1–13.

Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. 2009. Refining the Control Structure of Loops Using Static Analysis. In *Proceedings of the Seventh ACM International Conference on Embedded Software (EMSOFT '09)*. ACM, New York, NY, USA, 49–58. https://doi.org/10.1145/1629335.1629343

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. 2006. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 73–85.

Thomas Ball, Todd Millstein, and Sriram K. Rajamani. 2005. Polymorphic Predicate Abstraction. *ACM Trans. Program. Lang. Syst.* 27, 2 (March 2005), 314–343. https://doi.org/10.1145/1057387.1057391

Thomas Ball and Sriram K Rajamani. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*. Springer-Verlag, 103–122.

Nels E Beckman, Duri Kim, and Jonathan Aldrich. 2011. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming*. Springer, 2–26.

Nels E Beckman, Aditya V Nori, Sriram K Rajamani, Robert J Simmons, Sai Deep Tetali, and Aditya V Thakur. 2010. Proofs from tests. *IEEE Transactions on Software Engineering* 36, 4 (2010), 495–508.

Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. *ACM SIGPLAN Notices* 42, 10 (2007), 301–320.

Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. 2009. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*. Springer, 195–219.

Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. 2013. Tree interpolation in vampire. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 173–181.

Eric Bodden. 2010. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 5–14.

Eric Bodden and Laurie Hendren. 2012. The Clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer* 14, 3 (2012), 307–326.

Dario Carotenuto, Aniello Murano, and Adriano Peron. 2007. 2-visibly pushdown automata. In *International Conference on Developments in Language Theory*. Springer, 132–144.

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 30.

Feng Chen and Grigore Roşu. 2007. Mop: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 569–588. https://doi.org/10.1145/1297027.1297069

Hao Chen and David Wagner. 2002. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 235–244.

Noam Chomsky. 1959. On certain formal properties of grammars. *Information and control* 2, 2 (1959), 137–167.

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software*, Alastair Donaldson and David Parker (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 248–254.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)* 50, 5 (2003), 752–794.

John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps. 2019. Refinement of Path Expressions for Static Analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 45 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290358

Marcelo d'Amorim and Klaus Havelund. 2005. Event-based Runtime Verification of Java Programs. In *Proceedings of the Third International Workshop on Dynamic Analysis (WODA '05)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/1082983.1083249

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

Robert DeLine and Manuel Fähndrich. 2004. Typestates for objects. In *European Conference on Object-Oriented Programming*. Springer, 465–490.

Android Developers Documentation. 2020. https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed: 2020-07-03.

Javier Esparza, Antonın Kučera, and Stefan Schwoon. 2003. Model checking LTL with regular valuations for pushdown systems. *Information and Computation* 186, 2 (2003), 355–376.

Kostas Ferles, Jon Stephens, and Isil Dillig. 2020. Verifying Correct Usage of Context-Free API Protocols (Extended Version). arXiv:cs.PL/2010.09652

Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 9.

Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. 2002. Extended static checking for Java. *ACM Sigplan Notices* 37, 5 (2002), 234–245.

Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems*, Jacques Garrigue (Ed.). Springer International Publishing, Cham, 275–295.

Pierre Ganty, Rupak Majumdar, and Benjamin Monmege. 2010. Bounded Underapproximations. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 600–614.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 4 (2014), 12.

Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. Compositional May-must Program Analysis: Unleashing the Power of Alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/1706299.1706307

Sergey Grebenshchikov, Ashutosh Gupta, Nuno P Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. HSF (C): A software verifier based on Horn clauses. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 549–551.

Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. 2006. SYNERGY: A New Algorithm for Property Checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 117–127. https://doi.org/10.1145/1181775.1181790

Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow Refinement and Progress Invariants for Bound Analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 375–385. https://doi.org/10.1145/1542476.1542518

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.

Michael A Harrison, Ivan M Havel, and Amiram Yehudai. 1979. On equivalence of grammars through transformation trees. *Theoretical Computer Science* 9, 2 (1979), 173–205.

Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, et al. 2018. Ultimate Automizer and the Search for Perfect Interpolants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 447–451.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested interpolants. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 471–482. https://doi.org/10.1145/1706299.1706353

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004a. Abstractions from Proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 232–244. https://doi.org/10.1145/964001.964021

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. 2004b. Abstractions from proofs. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 232–244.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 58–70. https://doi.org/10.1145/503272.503279

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software verification with BLAST. In *International SPIN Workshop on Model Checking of Software*. Springer, 235–239.

John E Hopcroft. 2008. *Introduction to automata theory, languages, and computation*. Pearson Education India.

Graham Hughes and Tevfik Bultan. 2008. Interface grammars for modular software model checking. *IEEE Transactions on Software Engineering* 34, 5 (2008), 614–632.

Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 1427–1430.

Pallavi Joshi and Koushik Sen. 2008. Predictive typestate checking of multithreaded java programs. In *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 288–296.

Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 352–358.

Allen J Korenjak and John E Hopcroft. 1966. Simple deterministic languages. In *7th Annual Symposium on Switching and Automata Theory (swat 1966)*. IEEE, 36–46.

Patrick Lam, Viktor Kuncak, and Martin Rinard. 2004. Generalized typestate checking using set interfaces and pluggable analyses. *ACM SIGPLAN Notices* 39, 3 (2004), 46–55.

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast Graph Simplification for Interleaved Dyck-Reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 780–793. https://doi.org/10.1145/3385412.3386021

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. 2012. Language-Theoretic Abstraction Refinement. In *Fundamental Approaches to Software Engineering*, Juan de Lara and Andrea Zisman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 362–376.

Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating grammar comparison. In *Acm Sigplan Notices*, Vol. 50. ACM, 183–200.

Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 365–383. https://doi.org/10.1145/1094811.1094840

Kenneth L McMillan. 2005. Applications of Craig interpolants in model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1–12.

Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*. Springer, 123–136.

Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. 2010. Efficient monitoring of parametric context-free patterns. *Automated Software Engineering* 17, 2 (2010), 149–180.

Tmima Olshansky and Amir Pnueli. 1977. A direct algorithm for checking equivalence of LL (k) grammars. *Theoretical Computer Science* 4, 3 (1977), 321–349.

Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. 2012a. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 925–935.

Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. 2012b. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 925–935.

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

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 49–61.

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 387–400.

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/1250734.1250748

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 59–76.

Robert E Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* 1 (1986), 157–171.

Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 83–95. https://doi.org/10.1145/2676726.2676997

Aditya Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas Reps. 2010. Directed Proof Generation for Machine Code. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 288–305. https://doi.org/10.1007/978-3-642-14295-6_27

S. L. Torre, P. Madhusudan, and G. Parlato. 2007. A Robust Class of Context-Sensitive Languages. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 161–170. https://doi.org/10.1109/LICS.2007.9

Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. 2013. On Multi-stack Visibly Pushdown Languages.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 13.

Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, 439–449.

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*. Springer, 98–122.

Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic verification of regular properties. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 871–881.

Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 435–446.

Qirun Zhang and Zhendong Su. 2017. Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3009837.3009848